

The script-writer's dream: How to write great SQL in your own language, and be sure it will succeed (Tech Report)

Ezra e. k. Cooper

June 16, 2009

Abstract Accessing a relational database from program code is a bugbear of many programmers—among them those who write “scripts” or web applications. The relational model frequently mismatches with the natural data structures of the application, and it is necessary to write queries in the specialized and peculiar language called SQL (often pronounced “sequel”) rather than using native iteration and conditional constructs.

Here we will see how to translate ordinary higher-order functional program code into SQL queries, and give a static analysis to determine the translatability of designated expressions. Somewhat surprisingly, we show that any appropriately-typed pure functional expression translates to a single SQL query, albeit for a strong notion of “purity.” Thus, unlike in Hollywood where a script-writer can never be sure a movie sequel will succeed at the box office, we show how to be sure that your SQL—written in your own language—will succeed (in being translated).

Introduction

A programmer's interface to a relational database—or any persistent data—is rarely comfortable. The SQL language is quite powerful for expressing queries on relational data, but its syntax is cumbersome and ill-matched to ordinary general-purpose programming languages. Common practices for bridging the distance are either unsafe, or inflexible, or use a different data model, giving rise to an impedance mismatch problem. Programmers often form SQL queries simply by concatenating strings at runtime (“string interpolation”); this is risky, as it becomes easy to make malformed or undesired queries. Sometimes programmers develop a library of “prepared statements” which are

parameterized only in controlled ways; this is safer but inflexible, requiring the programmer to dip into the library each time a prepared statement needs to be adjusted. Object–Relational mappings (ORMs) provide a safe and flexible object-oriented interface to data, but moving between the object and relational models can give rise to subtle problems, for example when object identity is relevant on the object side, or complex join queries that are inexpressible on the object side are needed.

All common approaches fall short in abstraction: none allows applying abstracted query fragments, such as “where” conditions, in multiple contexts. And all of them bring their own query language, with its own peculiar syntax (this language may be SQL itself or the language comprised by the ORM library).

This paper shows how to integrate general-purpose functional programming languages with SQL, thus providing more abstraction for writing queries and reducing the impedance mismatch. The programmer can write queries in her “native” programming language, using the same constructs as she normally uses to iterate over collection structures. In the body of an iteration, she can create data structures that have no immediate representation in SQL, as long as the result structure is representable. And, an annotation applied to the source allows her to determine at compile-time whether important code fragments will in fact become SQL queries.

Usage An example use of the system might be as follows. Alice writes a program that iterates over *table handle* values, which represent tables in a connected database. Alice’s compiler may choose to implement the iteration naively, by fetching whole tables and processing them like any other collection data; or it may discover that the table-handle reference appears within a larger expression that is equivalent to a query, and it may implement the larger expression directly as an SQL query. But Alice believes that a certain expression is equivalent to a query, and wants to be sure that at least this expression is implemented that way. So she applies an annotation to the expression, wrapping it in “query brackets.” Now when she compiles the program, the compiler makes a decision about whether it can implement that expression entirely as an SQL query. If so, it accepts the expression, and Alice knows that it will run that way. If not, it gives a compile-time error, perhaps explaining why the expression cannot certainly be translated to a query.

(This story gives the compiler freedom to execute additional expressions as queries, besides the bracketed ones. Alternatively, the language could be designed to create queries only when query brackets are used. The choice is up to the language designer.)

Next, after writing a function that performs a query with query brackets, Alice decides that one of the conditions used in the query should in fact be a parameter, so that clients of her API could pass any predicate they like. She simply replaces the direct comparison with a call to a given first-class function. Because the call falls within query brackets, the compiler will ensure that every call to her function passes a predicate that can be translated to SQL.

Example

Suppose Alice runs a local baseball league. First, she wants a list of the players with age at least 16. She might write this function (these examples use the syntax of Links, which is a general-purpose language but is specially adapted to look like a query language):

```
fun overAgePlayers() {
  query {
    for (p <- players)
    where (p.age > 16)
      [(name = p.name)] }
}
```

(The `for (x <- xs) e` construct is a *bag comprehension*, and works as follows: for each element in the collection obtained by evaluating `xs`, it evaluates `e` with `x` bound to that element, producing a new collection; the value of the comprehension is the union of all the collections produced. The construct `where (e1) e2` is simply shorthand for `if e1 then e2 else []`.) The compiler can deduce that this expression is in fact equivalent to an SQL query (it is *queryizable*), so it accepts the function. However, the following code would give a compiler error:

```
fun overAgePlayersReversed() {
  query {
    for (p <- players)
    where (p.age > 16)
      [(name = reverse(p.name))] } # ERROR!
}
```

This is because the *reverse* function has no SQL equivalent, and so no query is equivalent to this expression.

Now, it takes nine players to make a baseball team, but some “teams” in Alice’s league are short of players. One way to find them is to collect, for each team, a team roster (list of players) and then filter for those with a roster of size at least nine. She needs to generate a list of players that belong to a “short” team, to inform them that they won’t be able to play this season. She writes the following code:

```

fun teamRosters() {
  for (t <- teams)
    [(name = t.name,
      roster = for (p <- players)
        where (p.team == t.name)
          [(playerName=p.name)]))]
}
fun usablePlayers() {
  query {
    for (t <- teamRosters())
      where (length(t.roster) >= 9)
        t.roster
  }
}

```

Note the lack of brackets [] in the body of the final comprehension: since the `for`-comprehension takes the *union* of collections produced by the body, the last comprehension takes the union of the rosters satisfying the condition. This expression is equivalent to an SQL query, although not in a direct way, since it uses an intermediate data structure that is nested (`teamRosters` has type `[(name:String, roster:[(playerName:String)])]`), and this is not supported by SQL. But since the final result is flat, our analysis accepts the query-bracketed expression and translates it into an equivalent SQL query, such as this one:

```

select p.name as playerName
  from players as p, teams as t
 where (select count(*) from players as p2
        where p2.team = t.name) < 9)

```

Also note that factoring out the part of the query which returns the complete bag of rosters poses no problem: the query translator will simply inline the function and produce a single query.

Suppose now that Alice wishes to abstract the query condition on teams. That is, she wishes to write a function which accepts as argument a roster predicate, and produces a list of the player records belonging to those teams satisfying the predicate. If she writes the following code, the query translator will still produce, in each invocation, a single SQL query:

```

fun playersBySelectedTeams(pred) {
  query {
    for (t <- teamRosters())
      where (pred(t.roster))
        t.roster
  }
}

```

The query translator will ensure that any argument passed as `pred` is itself a queryziable function. If any call site tries to pass a non-queryziable predicate, it produces a compiler

error.

This type of abstraction is particularly difficult to achieve in SQL, since the team rosters themselves cannot be explicitly constructed in SQL as part of a query. SQL places restrictions on how subqueries can be used (For example, in various contexts they must return just one column, just one row, or both) and has a non-orthogonal syntax so that the subquery itself must be changed depending on how it is used.

For example, suppose we want to form a sub-league of “senior” teams: teams with enough players of age 15. We might define the following predicates:

```
fun fullTeam(list) { length(list) >= 9 }
fun seniorPlayers(list) { for (x <- xs) where (x.age >= 15) [x] }
```

and use them as follows:

```
playersBySelectedTeams(fun(x) { fullTeam(eligiblePlayers(x)) } )
```

Here we have freely used *length* and a filtering comprehension, despite the fact that in SQL these are represented very differently.

```
select p.name as playerName
  from players as p, teams as t
  where (select count(*) from players as p2
         where p2.team = t.name and p2.age >=15) >= 9)
```

The application of `count` needs to be placed within the subquery, while the comparison `>= 9` is placed outside of it. Also the condition on `p2.age` must be placed within the `where` clause of the subquery. It would not be easy to write a “template” SQL query which would support all the queries as *playersBySelectedTeams* through string substitution.

How it works The recipe for compiling abstractable, higher-order language-integrated queries can be summarized as follows:

1. At compile time,
 - a) Check statically that query expressions have flat relation type.
 - b) Use a *type-and-effect system* to determine that annotated expressions are pure.
 - c) Tag the runtime representation of each first-class function with its source code.
2. At runtime, to execute an expression via SQL,
 - a) insert the values for any free variables (inserting the source code, in the case of functions) and
 - b) “evaluate” it to eliminate intermediate structures (that is, functions and nested data structures). This produces a normal form directly translatable to SQL.

Microsoft’s LINQ system allows abstraction of the kind shown in the last examples, provided that the predicates are defined at a special type (the type of *expressions*). Because of the special type, these predicates cannot directly be reused as ordinary functions, and expression composition is not supported in queries, so the last example would require explicitly rewriting the composed function `fun(x) { fullTeam(eligiblePlayers(x))}` as a new function. This paper shows how such a system might support expression composition.

This paper Here we show that any closed, “pure” expression expressible in a typical impure functional programming language, as long as it has flat relation type (a bag of records of base values), can be translated to an equivalent single SQL query. We understand “pure” to include the abjuration of recursive computations and non-SQL-expressible functions. (But recursion is only disallowed from SQL expressions and not from the language as a whole.) This set of expressions is enough to express any query in a significant fragment of SQL.

Furthermore, given an open expression, we can statically detect whether for all well-typed substitutions on those free variables the expression can be translated to an SQL query. The type system will ensure that only database-friendly values are passed for those variables.

We also propose a trivial language extension allowing a programmer to express the intent that an expression should translate to a single SQL query, or fail statically if this cannot be guaranteed.

Road map The next sections (1) define the core source language and its static analysis, (2) translate it into SQL, (3) characterize the correctness of the algorithm, and (4) extend it with recursion, while disallowing recursion within query expressions.

1 The Source Language

We define a language which resembles the core of an ordinary impure functional programming language, and is also a conservative extension of the (higher-order) Nested

Relational Calculus with side-effects and a “query” annotation. Its grammar is as follows:

(terms)	B, L, M, N	$::=$	for $(x \leftarrow L) M$
			if B then M else N
			table $s : T$
			$[M] \mid [] \mid M \uplus N$
			$(\overrightarrow{l = \vec{M}}) \mid M.l$
			$\lambda x.N \mid LM \mid x \mid c$
			$\oplus(\vec{M})$
			empty (M)
			query M
(primitives)			\oplus
(table names)			s, t
(field names)			l
(types)			$T ::= o \mid (\overrightarrow{l : T}) \mid [T] \mid S \xrightarrow{e} T$
(base types)			$o ::= \mathbf{bool} \mid \mathbf{int} \mid \mathbf{string}$
(atomic effects)			$E ::= \mathbf{noqy} \mid \dots$
(effect sets)			e a set of atomic effects

The terms $[M]$, $[]$ and $M \uplus N$ represent bag (multiset) operations: singleton construction, the empty bag, and bag-union.

We consider effects abstractly; E ranges over some arbitrary set of effects, which includes at least an effect **noqy** and may include other runtime actions such as I/O or reference-cell mutations. Every effect should represent some kind of runtime behavior that has no SQL equivalent; we use the distinguished effect **noqy** to mark nondatabasable operations when no other effect presents itself.

The form $\oplus(\vec{M})$ denotes the full application of a primitive \oplus to arguments \vec{M} . An operation cannot appear without being applied.

Every primitive \oplus either has an SQL equivalent \oplus_{sql} or carries an effect annotation (perhaps the catch-all **noqy**). The SQL equivalent is a macro which expands to some combination of primitives available in SQL. Thus primitives need not be in one-to-one correspondance with real SQL operations. We also insist that primitives have basic argument types and basic result type, or else have an effect annotation.

NRC Comparison Compare the given language with the Nested Relational Calculus (NRC) as given by Wong [Wong, 1996]:

This work			NRC		
LM	$\lambda x.N$	x	LM	$\lambda x.M$	x
	if B then M else N			if B then M else N	
	empty M			empty M	
	c			c	
	for $(x \leftarrow L) M$			$\bigcup\{M \mid x \in L\}$	
$[\]$	$[M]$	$M \uplus N$	$\{\}$	$\{M\}$	$M \cup N$
	table $s : T$		x		
	$(\overrightarrow{l = \overrightarrow{M}})$	$M.l$	$()$	(L, M)	$\pi_1 \quad \pi_2$
	$\oplus(\overrightarrow{M})$				
	query M			$M = N$	

The two languages are nearly the same. Some apparent differences are only notational. NRC's comprehension form $\bigcup\{M \mid x \in L\}$ is identical to ours, $\text{for } (x \leftarrow L) M$. NRC lists the unit value $()$ separately while we treat it as a 0-field record. The NRC literature uses set notation, $\{\}$, $\{M\}$, and $M \cup N$, but they can refer to any of the extended monads for bags, sets or lists. (Here we treat only bags.) We write table handles explicitly as **table** $s : T$, with a name s and required type annotation T , which facilitates local translation, while NRC uses free variables to refer to tables.

A few differences go deeper than notation. NRC uses tuples while we use records, a mild generalization. We admit an arbitrary set of primitives \oplus while NRC, at its core, includes no operations; instead they are normally treated as extensions. This formulation of NRC includes an equality test at each type; the equality at base types is treated as a primitive (ranged by \oplus) in our calculus. Finally, our language extends NRC with the databasability annotation **query** M .

Type-and-effect system The type-and-effect system is given in Figure 1. It is close to a standard type-and-effect system along the lines of Talpin and Jouvelot [1992]. The system permits no recursion, and thus is analogous to simply-typed λ -calculus. We add recursion later using a fixpoint operator.

The type of each constant c is given by T_c , which must be a base type. Constant values at complex type can, of course, be constructed explicitly. Each primitive has a given type, denoted by a judgment $\oplus : S_1 \times \dots \times S_n \xrightarrow{e} T$.

In keeping with the flat structure of SQL tables, we require that each table must have flat relation type: in **table** $s : T$ we require $T = [(\overrightarrow{l : o})]$ for some record type $(\overrightarrow{l : o})$.

An immediate type annotation is required on table expressions. This may seem a nuisance; in fact it is not strictly necessary, since an algorithm could infer the type at which the table is used. As a pragmatic matter, however, it provides a direct way for the programmer to check whether the usage type of a table agrees with the underlying table's schema type in the DBMS. Ensuring such agreement is beyond the scope of this

$$\begin{array}{c}
\Gamma \vdash c : T_c ! \emptyset \quad (\text{T-CONST}) \quad \Gamma \vdash M_l : T_l ! e_l \text{ for each pair } M_l, T_l \\
\text{such that } l = M_l \text{ in } \overrightarrow{l = M} \\
\Gamma, x : T \vdash x : T ! \emptyset \quad (\text{T-VAR}) \quad \text{and } l : T_l \text{ in } \overrightarrow{l : T} \\
\hline
\Gamma, x : S \vdash N : T ! e' \quad (\text{T-ABS}) \quad \Gamma \vdash (\overrightarrow{l = M}) : (\overrightarrow{l : T}) ! \bigcup_{l \in \vec{l}} e_l \\
\Gamma \vdash \lambda x. N : S \xrightarrow{e'} T ! \emptyset \quad (\text{T-REC}) \\
\hline
\Gamma \vdash L : S \xrightarrow{e} T ! e' \quad \Gamma \vdash M : S ! e'' \quad (\text{T-PROJECT}) \\
\hline
\Gamma \vdash LM : T ! e \cup e' \cup e'' \quad (\text{T-APP}) \\
\hline
\oplus : S_1 \times \dots \times S_n \xrightarrow{e} T \\
\Gamma \vdash M_i : S_i ! e_i \text{ for each } 1 \leq i \leq n \quad (\text{T-OP}) \\
\hline
\Gamma \vdash \oplus(\vec{M}) : T ! e \cup \bigcup_i e_i \\
\hline
\Gamma \vdash M : T ! \emptyset \\
T \text{ has the form } [\overrightarrow{l : \delta}] \quad (\text{T-DB}) \\
\hline
\Gamma \vdash \text{query } M : T ! \emptyset \\
\hline
T \text{ has the form } [\overrightarrow{l : \delta}] \quad (\text{T-TABLE}) \\
\hline
\Gamma \vdash (\text{table } t : T) : T ! \emptyset \\
\hline
\Gamma \vdash M : [S] ! e \\
\Gamma, x : S \vdash N : [T] ! e' \quad (\text{T-FOR}) \\
\hline
\Gamma \vdash \text{for } (x \leftarrow M) N : [T] ! e \cup e' \\
\hline
\Gamma \vdash M : T ! e \quad \Gamma \vdash N : [T] ! e' \\
\hline
\Gamma \vdash M.l : T_l ! e \quad (l : T_l) \in (\overrightarrow{l : T}) \\
\hline
\Gamma \vdash M.l : T_l ! e \quad (\text{T-NULL}) \\
\hline
\Gamma \vdash M : T ! e \\
\hline
\Gamma \vdash [M] : [T] ! e \quad (\text{T-SINGLETON}) \\
\hline
\Gamma \vdash M : [T] ! e \quad \Gamma \vdash N : [T] ! e' \\
\hline
\Gamma \vdash M \uplus N : [T] ! e \cup e' \quad (\text{T-UNION}) \\
\hline
\Gamma \vdash M : [T] ! e \\
\hline
\Gamma \vdash \text{empty}(M) : \text{bool} ! e \quad (\text{T-EMPTY}) \\
\hline
\Gamma \vdash L : \text{bool} ! e \\
\Gamma \vdash M : T ! e' \quad \Gamma \vdash N : T ! e'' \\
\hline
\Gamma \vdash \text{if } L \text{ then } M_1 \text{ else } M_2 : T ! e \cup e' \cup e'' \quad (\text{T-IF}) \\
\hline
\Gamma \vdash M : T ! e \quad e \subseteq e' \quad (\text{T-SUBSUMP}) \\
\hline
\Gamma \vdash M : T ! e'
\end{array}$$

Figure 1: Type-and-effect system.

theory; but at least this immediate type annotation permits a simple source-level check, where otherwise it would be necessary to run type inference first.

This type system is monomorphic, so for example each appearance of the empty bag $[]$ must be given some particular concrete type. Adding polymorphism presents some difficulties of its own and is left for future work.

2 Making Queries

To make queries from the source language, we will rewrite source terms to a sublanguage which embeds easily in our SQL subset.

We first examine the sublanguage and its relationship to our SQL fragment, then turn to the rewrite system.

SQL-like sublanguage Our chosen sublanguage is as follows:

$$\begin{array}{ll}
\text{(normal forms)} & V, U ::= V \uplus U \mid [] \mid F \\
\text{(comprehension NFs)} & F ::= \text{for } (x \leftarrow \text{table } s : T) F \mid Z \\
\text{(comprehension bodies)} & Z ::= \text{if } B \text{ then } Z \text{ else } [] \mid [R] \mid \text{table } s : T \\
\text{(row forms)} & R ::= \overrightarrow{(l = \vec{B})} \mid x \\
\text{(basic expressions)} & B ::= \text{if } B \text{ then } B' \text{ else } B'' \mid \text{empty}(V) \mid \\
& \quad \oplus(\vec{B}) \mid x.l \mid c
\end{array}$$

Observe that the “normal form” expressions ranged by V all have *relation type*: bag of record of base type, or $[(l : \vec{o})]$. Types of the form $\overrightarrow{(l : \vec{o})}$ are called “row types,” after the database rows that they represent.

SQL fragment Our SQL fragment is as follows:

$$\begin{array}{l}
Q, R ::= Q \text{ union all } R \mid S \\
S ::= \text{select } \overrightarrow{e} \text{ as } \vec{l} \text{ from } \overrightarrow{t} \text{ as } \vec{x} \text{ where } e \\
e ::= \text{case when } e \text{ then } e' \text{ else } e'' \text{ end} \mid \\
\quad c \mid x.l \mid e \wedge e' \mid \neg e \mid \text{exists}(Q) \mid \oplus(\vec{e})
\end{array}$$

This includes all unions of queries on an inner join of zero or more tables, with result and query conditions taken from some given algebra of operations, including field projection, boolean conjunction, negation, the `exists` operator, and conditionals `case . . . end`.

SQL translation Now the type-directed function $\llbracket - \rrbracket$ translates each closed term in the sublanguage directly into a query:

$$\begin{aligned}
\llbracket V \uplus U \rrbracket &= \llbracket V \rrbracket \text{ union all } \llbracket U \rrbracket \\
\llbracket [] : [(l : \vec{T})] \rrbracket &= \text{select null as } \vec{l} \text{ from } \emptyset \text{ where false} \\
\llbracket \text{for } (x \leftarrow \text{table } s : T) F \rrbracket &= \text{select } \vec{e} \text{ as } \vec{l} \text{ from } s \text{ as } x, \vec{t} \text{ as } \vec{y} \text{ where } B \\
&\quad \text{where } (\text{select } \vec{e} \text{ as } \vec{l} \text{ from } \vec{t} \text{ as } \vec{y} \text{ where } B) = \llbracket F \rrbracket \\
\llbracket \text{if } B \text{ then } Z \text{ else } [] \rrbracket &= \text{select } \vec{e} \text{ as } \vec{l} \text{ from } \vec{t} \text{ where } B' \wedge \llbracket B \rrbracket \\
&\quad \text{where } (\text{select } \vec{e} \text{ as } \vec{l} \text{ from } \vec{t} \text{ where } B') = \llbracket Z \rrbracket \\
\llbracket \text{table } s : [(l : \vec{o})] \rrbracket &= \text{select } \vec{s.l} \text{ as } \vec{l} \text{ from } s \text{ where true} \\
\llbracket [(l = \vec{B})] \rrbracket &= \text{select } \llbracket B \rrbracket \text{ as } \vec{l} \text{ from } \emptyset \text{ where true} \\
\llbracket \text{if } B \text{ then } B' \text{ else } B'' \rrbracket &= \text{case when } \llbracket B \rrbracket \text{ then } \llbracket B' \rrbracket \text{ else } \llbracket B'' \rrbracket \text{ end} \\
\llbracket \text{empty}(V) \rrbracket &= \neg \text{exists}(\llbracket V \rrbracket) \\
\llbracket \oplus(\vec{B}) \rrbracket &= \oplus_{\text{sql}}(\llbracket \vec{B} \rrbracket) \\
\llbracket x.l \rrbracket &= x.l \\
\llbracket x \rrbracket &= x.* \\
\llbracket c \rrbracket &= c
\end{aligned}$$

(A fine point: SQL has no way of selecting an empty set of result columns in a `select` clause; to translate $[(\)]$ we need to offer some dummy value, or `*`, as the result column. An implementation can supply any such dummy value, remembering to ignore the columns that the database actually returns.)

Because we assume that all bound variables in the source program are distinct, there will be no clashes among the table aliases (the identifiers following the `as` keywords) produced by this translation.

Rewrite rules The translation of source terms into the SQL-isomorphic sublanguage is given as a strongly-normalizing rewrite system (Figure 2). We write $M[L/x]$ for the substitution of the term L for the free variable x in the term M .

A few rules beg explanation. The `EMPTY-FLATTEN` rule ensures that no query inside `empty` can have a problematic type; we transform each row of M to the unit value, which preserves emptiness. The `IF-SPLIT` rule transforms a choice between two bags—which has no direct analogue in SQL—into the union of two oppositely-guarded bags. Rules like `APP-IF` are common in higher-order strongly-normalizing rewrite systems: they push the deconstructors $-l$ and $-M$ past an interposing form, in order to bring them together with corresponding constructors $(l = \vec{M})$ and $\lambda x.N$ thus exposing β -reductions. `IF-RECORD` is not needed for strong normalization, but only to produce normal forms that are structurally close to SQL. So `IF-RECORD` forces conditional choices to be made not on a whole row but on each field in the row.

$$\begin{array}{l}
\text{for } (x \leftarrow [M]) N : T \rightsquigarrow N[M/x] \quad (\text{FOR-}\beta) \\
(\lambda x.N)M : T \rightsquigarrow N[M/x] \quad (\text{ABS-}\beta) \\
(\overrightarrow{l = M}).l_i : T \rightsquigarrow M_i \quad (\text{RECORD-}\beta) \\
\text{for } (x \leftarrow []) M : T \rightsquigarrow [] \quad (\text{FOR-ZERO-SRC}) \\
\text{for } (x \leftarrow N) [] : T \rightsquigarrow [] \quad (\text{FOR-ZERO-BODY}) \\
\text{for } (x \leftarrow \text{for } (y \leftarrow L) M) N : T \rightsquigarrow \text{for } (y \leftarrow L) (\text{for } (x \leftarrow M) N) \quad \text{if } y \notin \text{FV}(N) \quad (\text{FOR-ASSOC}) \\
\text{for } (x \leftarrow M_1 \uplus M_2) N : T \rightsquigarrow \text{for } (x \leftarrow M_1) N \uplus \text{for } (x \leftarrow M_2) N \quad (\text{FOR-UNION-SRC}) \\
\text{for } (x \leftarrow M) (N_1 \uplus N_2) : T \rightsquigarrow \text{for } (x \leftarrow M) N_1 \uplus \text{for } (x \leftarrow M) N_2 \quad (\text{FOR-UNION-BODY}) \\
\text{for } (x \leftarrow \text{if } B \text{ then } M \text{ else } []) N : T \rightsquigarrow \text{if } B \text{ then } (\text{for } (x \leftarrow M) N) \text{ else } [] \quad (\text{FOR-IF-SRC}) \\
(\text{if } B \text{ then } L \text{ else } L')M : T \rightsquigarrow \text{if } B \text{ then } LM \text{ else } L'M \quad (\text{APP-IF}) \\
\text{if } B \text{ then } M \text{ else } N : (\overrightarrow{l : \hat{T}}) \rightsquigarrow (\overrightarrow{l = \hat{L}}) \quad (\text{IF-RECORD}) \\
\text{where } L_l = \text{if } B \text{ then } M.l \text{ else } N.l \text{ for each } l \in \vec{l} \\
\text{if } B \text{ then } M \text{ else } N : [T] \rightsquigarrow \text{if } B \text{ then } M \text{ else } [] \quad (\text{IF-SPLIT}) \\
\quad \uplus \text{if } \neg B \text{ then } N \text{ else } [] \quad \text{if } N \neq [] \\
\text{if } B \text{ then } [] \text{ else } [] : T \rightsquigarrow [] \quad (\text{IF-ZERO}) \\
\text{if } B \text{ then } (\text{for } (x \leftarrow M) N) \text{ else } [] : T \rightsquigarrow \text{for } (x \leftarrow M) (\text{if } B \text{ then } N \text{ else } []) \quad (\text{IF-FOR}) \\
\text{if } B \text{ then } M \uplus N \text{ else } [] : T \rightsquigarrow \text{if } B \text{ then } M \text{ else } [] \quad (\text{IF-UNION}) \\
\quad \uplus \text{if } B \text{ then } N \text{ else } [] \\
\text{empty}(M) : T \rightsquigarrow \text{empty}(\text{for } (x \leftarrow M) [()]) \quad \text{if } M \text{ is not relation-typed} \\
\quad (\text{EMPTY-FLATTEN}) \\
\text{query } M : T \rightsquigarrow M \quad (\text{IGNORE-DB})
\end{array}$$

Figure 2: The rewrite system for normalizing source-language terms.

Several rewrite rules may seem unnecessary if we are permitted SQL subqueries. For example, why employ the FOR-ASSOC rule if we can write an SQL query that uses a nested SQL select statement in its from clause? After all, we could more directly translate the expression

$$\text{for } (y \leftarrow \text{for } (x \leftarrow \text{table } s : T) [(b = x.a)]) [(c = y.b)]$$

into SQL as follows:

$$\text{select } y.b \text{ as } c \text{ from } (\text{select } x.a \text{ as } b \text{ from } s \text{ as } x) \text{ as } y.$$

In this case, the nested comprehension became a nested subquery. So why include FOR-ASSOC?

The answer is that such rules are critical to the unnesting. Consider this query which creates an intermediate result of nested bag-of-bag type, not an SQL-representable type:

$$\text{for } (y \leftarrow \text{for } (x \leftarrow \text{table } s) [[x]]) y$$

The expression rewrites using the FOR-ASSOC rule:

$$\begin{aligned} \text{for } (y \leftarrow (\text{for } (x \leftarrow \text{table } s) [[x]]) y) &\rightsquigarrow && \text{(FOR-ASSOC)} \\ \text{for } (x \leftarrow \text{table } s) (\text{for } (y \leftarrow [[x]]) y) &\rightsquigarrow && \text{(\beta-FOR)} \\ &\text{for } (x \leftarrow \text{table } s) [x] \end{aligned}$$

and now the expression is unnested. The FOR-ASSOC rule thus exposes β reductions which themselves eliminate constructor/destructor pairs and hence reduce the types of intermediate values.

This rewrite system is strongly normalizing for well-typed terms, as we will see, so we can use it in translating our language to SQL without fear that it will run forever.

This, then, is the translation from source to SQL, defined on terms M for which $\text{query}(M)$ is well-typed: normalize the term using the rewrite system and then translate it to SQL using the $\llbracket - \rrbracket$ function given above.

3 Correctness

Soundness

The system is useless if rewriting changes the behavior of terms. In particular, rewriting should not generate new side-effects. This section shows that reduction preserves types and purity.

Normally, in an effect system with an operational semantics, the reductions in that semantics preserve types and effects. But since here the rewriting strategy is not specified (rewrites can be made anywhere in a term), types and effects are not always preserved. Indeed, the rules are only sound for pure terms, but this is all we plan to use them for—so we'll show that purity is preserved by reduction.

The problem is that substitution can modify the effects captured by a function type. For example suppose M has typing $\vdash M : S ! e$ with e a non-empty effect set. We can rewrite $(\lambda x. \lambda y. x)M \rightsquigarrow \lambda y. M$. The redex types as $\vdash (\lambda x. \lambda y. x)M : T \xrightarrow{\emptyset} S ! e$ while the reduct types as $\vdash \lambda y. M : T \xrightarrow{e} S ! \emptyset$: the reduction relocates the effect, disturbing the type.

To deal with this problem we introduce a function that pushes an effect down onto all of the function arrows appearing in a type, and a function that completes a type by pushing down a given effect as well and recursively pushes all the effects that appear down into the subordinate types.

Definition. Define a function *push* which pushes effects into every arrow in a type:

$$\begin{aligned} \text{push } e' (S \xrightarrow{e} T) &= \text{push } e' S \xrightarrow{e \cup e'} \text{push } e' T \\ \text{push } e' [T] &= [\text{push } e' T] \\ \text{push } e' (\overrightarrow{l : T}) &= \overrightarrow{(l : (\text{push } e' T))} \\ \text{push } e' o &= o \end{aligned}$$

And a function *complete* which pushes effects into every arrow in a type and propagates effects in the type downwards:

$$\begin{aligned} \text{complete } e' (S \xrightarrow{e} T) &= \text{complete } (e \cup e') S \xrightarrow{e \cup e'} \text{complete } (e \cup e') T \\ \text{complete } e' [T] &= [\text{complete } e' T] \\ \text{complete } e' (\overrightarrow{l : T}) &= \overrightarrow{(l : (\text{complete } e' T))} \\ \text{complete } e' o &= o \end{aligned}$$

Define lifted versions of both of these functions for typing contexts by applying them to each type in the context.

Now, a typing can be “weakened” so that we can assign the term a type which is pushed or completed by any effect.

Lemma 1. If $\Gamma \vdash M : T ! e$ then $\Gamma' \vdash M : T' ! e \cup e'$ where $\Gamma' = \text{complete } e' \Gamma$ and $T' = \text{complete } e' T$.

Proof. By induction on the derivation $\Gamma \vdash M : T ! e$. Showing the most interesting cases:

- $\Gamma \vdash x : T ! \emptyset$. Then $x : T \in \Gamma$ and then $x : T' \in \Gamma'$ where $T' = \text{complete } e' T$. So $\Gamma' \vdash x : T' ! \emptyset$. And T-EFF-WEAKENING gives $\Gamma' \vdash x : T' ! \emptyset \cup e'$ as needed.
- $\Gamma \vdash LM : T ! e$. Then $\Gamma \vdash L : S \xrightarrow{e_f} T ! e_L$ and $\Gamma \vdash M : S ! e_M$ with $e = e_f \cup e_L \cup e_M$.

By IH, $\Gamma' \vdash L : S' \xrightarrow{e'_f} T' ! e_L \cup e'$ and $\Gamma' \vdash M : S' ! e_M \cup e'$ where $\Gamma' = \text{complete } e' \Gamma$, $e'_f = e_f \cup e'$, $S' = \text{complete } (e' \cup e_f) S$ and $T' = \text{complete } (e' \cup e_f) T$.

And so by T-APP, $\Gamma' \vdash LM : T' ! e_L \cup e_M \cup e_f \cup e'$ as needed.

- $\Gamma \vdash \lambda x.N : T ! \emptyset$. Let $T = T_1 \xrightarrow{e_N} T_2$.

So $\Gamma', x : T_1 \vdash N : T_2 ! e_N$. By IH, $\Gamma', x : T'_1 \vdash N : T'_2 ! e_N \cup e'$ where $\Gamma' = \text{complete } e' \Gamma$, $T'_1 = \text{complete } (e_N \cup e') T_1$, $T'_2 = \text{complete } (e_N \cup e') T_2$.

By T-ABS, $\Gamma' \vdash \lambda x.N : T'_1 \xrightarrow{e_N \cup e'} T'_2 ! \emptyset$.

And by T-EFF-WEAKENING, $\Gamma' \vdash \lambda x.N : T'_1 \xrightarrow{e_N \cup e'} T'_2 ! e_N \cup e'$.

Also $T'_1 \xrightarrow{e_N \cup e'} T'_2 = \text{complete } e' (T_1 \xrightarrow{e_N} T_2)$. \square

Lemma 2. If $\Gamma \vdash M : T ! e$ then $\Gamma' \vdash M : T' ! e \cup e'$ where $\Gamma' = \text{push } e' \Gamma$ and $T' = \text{push } e' T$.

Proof omitted. (Similar to Lemma 1)

Substitution of one effectful term into another gives us a term which might have the former's effect on any of its arrows; conservatively, we can derive the “pushed” effect.

Lemma 3. From $\Gamma, x : S \vdash N : T ! e_N$

and $\Gamma \vdash P : S ! e_P$

we get $\Gamma' \vdash N[P/x] : T' ! e_N \cup e_P$

where $\Gamma' = \text{push } e_P \Gamma$ and $T' = \text{push } e_P T$.

Proof. By induction on the derivation of $\Gamma, x : S \vdash N : T ! e_N$.

- T-VAR.

Case $N = x$. The typing is $\Gamma, x : S \vdash x : T ! \emptyset$ with $e_N = \emptyset$. Here $S = T$. So we have $\Gamma \vdash P : T ! e_P$. By Lemma 2, $\Gamma' \vdash P : T' ! e_P$ where $\Gamma' \vdash \text{complete } e_P \Gamma$ and $T' = \text{complete } e_P T$ as needed.

Case $N = y$. The typing is $\Gamma, x : S \vdash y : T ! \emptyset$ with $e_N = \emptyset$. Now $N[P/x] = y$ and so the typing of P becomes $\Gamma \vdash y : T ! e_P$. By Lemma 2, $\Gamma' \vdash y : T' ! e_P$.

- T-APP.

The typing is

$$\frac{\Gamma, x : S \vdash L : U \xrightarrow{e_f} T ! e_L \quad \Gamma, x : S \vdash M : U ! e_M}{\Gamma, x : S \vdash LM : T ! e_N}$$

with $e_N = e_f \cup e_L \cup e_M$.

By IH,

$$\Gamma' \vdash L[P/x] : U' \xrightarrow{e'_f} T' ! e'_L \quad \text{and} \quad \Gamma' \vdash M[P/x] : U' ! e'_M$$

with $U' = \text{push } e_P U$, $T' = \text{push } e_P T$, $e'_f = e_f \cup e_P$, $e'_L = e_L \cup e_P$, $e'_M = e_M \cup e_P$. Now by T-APP, $\Gamma' \vdash (LM)[P/x] : T' ! e'_f \cup e'_L \cup e'_M = e_P \cup e_N$.

- T-ABS. If the term is $\lambda x.N'$ then the substitution is a no-op and the result follows from Lemma 2 and T-EFF-WEAKENING. So assume it is $\lambda y.N'$ with $y \neq x$.

The typing is

$$\frac{\Gamma, x : S, y : T_1 \vdash N' : T_2 ! e_{N'}}{\Gamma, x : S \vdash \lambda y.N' : T_1 \xrightarrow{e_{N'}} T_2 ! \emptyset}$$

with $T = T_1 \xrightarrow{e_{N'}} T_2$.

By IH, $\Gamma', y : T'_1 \vdash N[P/x] : T'_2 ! e'_{N'} \cup e_P$ with $T'_2 = \text{push } e_P T_2$ and $(\Gamma', y : T'_1) = \text{push } e_P (\Gamma, y : T_1)$ so $T'_1 = \text{push } e_P T_1$.

Then by T-ABS, $\Gamma' \vdash \lambda y.N' : T'_1 \xrightarrow{e_{N'} \cup e_P} T'_2 ! \emptyset$ with $T'_1 \xrightarrow{e_{N'}} T'_2 = \text{push } e_P (T_1 \xrightarrow{e_{N'}} T_2)$ and by T-EFF-WEAKENING, $\Gamma' \vdash \lambda y.N' : T'_1 \xrightarrow{e_{N'} \cup e_P} T'_2 ! e_P$ as needed.

- T-EFF-WEAKENING.

The typing is

$$\frac{\Gamma, x : S \vdash N : T ! e_1}{\Gamma, x : S \vdash N : T ! e_1 \cup e_2}$$

By IH, $\Gamma' \vdash M : T' ! e_1 \cup e_P$ with $\Gamma' = \text{push } e_P \Gamma$ and $T' = \text{push } e_P T$. Then by T-EFF-WEAKENING, $\Gamma' \vdash M : T' ! e_1 \cup e_2 \cup e_P$ as needed. \square

Completion commutes with union in the effect argument.

Lemma 4. For any e, e' and T : *complete* e (*complete* $e' T$) = *complete* $(e \cup e') T$ and *push* e (*push* $e' T$) = *push* $(e \cup e') T$.

Proof. By induction on the type. \square

Completion by an effect subsumes pushing by the same effect.

Lemma 5. For any e and T , *complete* e (*push* $e T$) = *complete* $e T$.

Proof omitted.

And finally, any reduction from a term of type T produces a term typable at the completion of T through the starting term's own effect.

Lemma 6. If $\Gamma \vdash M : T ! e$ and $M \rightsquigarrow M'$ then $\Gamma' \vdash M' : T' ! e$ with $\Gamma' = \text{complete } e \Gamma$ and $T' = \text{complete } e T$.

Proof. By induction on M . Take cases on the reduction. Showing the most interesting cases.

- $LM \rightsquigarrow LM'$.

The typing is

$$\frac{\Gamma \vdash L : S \xrightarrow{e_f} T ! e_L \quad \Gamma \vdash M : S ! e_M}{\Gamma \vdash LM : T ! e = e_f \cup e_L \cup e_M}$$

By IH, $\Gamma_M \vdash M' : S' ! e_M$ where $\Gamma_M = \text{complete } e_M \Gamma$ and $S' = \text{complete } e_M S$.

Using Lemmas 1 and 4, we can generalize this to $\Gamma' \vdash M' : S'' ! e$ where $\Gamma' = \text{complete } e \Gamma$ and $S'' = \text{complete } e S$.

Similarly, $\Gamma' \vdash L : S'' \xrightarrow{e_f \cup e} T'' ! e_L$ where $T'' = \text{complete } e T$.

And so by T-APP, $\Gamma' \vdash LM' : T'' ! e$.

- $LM \rightsquigarrow L'M$.

The typing is

$$\frac{\Gamma \vdash L : S \xrightarrow{e_f} T ! e_L \quad \Gamma \vdash M : S ! e_M}{\Gamma \vdash LM : T ! e = e_f \cup e_L \cup e_M}$$

By IH, $\Gamma_L \vdash L' : S' \xrightarrow{e'_f} T' ! e_L$ where $\Gamma_L = \text{complete } e_L \Gamma$ and $S' \xrightarrow{e'_f} T' = \text{complete } e_L (S \xrightarrow{e_f} T)$ so $e'_f = e_f \cup e_L$, $S' = \text{complete } (e_f \cup e_L) S$ and $T' = \text{complete } (e_f \cup e_L) T$.

Using Lemmas 1 and 4, we can generalize this to $\Gamma' \vdash L' : S'' \xrightarrow{e''_f} T'' ! e$ where $\Gamma' \vdash \text{complete } e \Gamma$, $S'' = \text{complete } e S$, $T'' = \text{complete } e T$ and $e''_f = e \cup e_f = e$.

Similarly, $\Gamma' \vdash M : S'' ! e_M$

And so by T-APP, $\Gamma' \vdash L'M : T'' ! e''_f \cup e \cup e_M = e$.

- ABS- β . $(\lambda x.N)M \rightsquigarrow N[M/x]$. The typing is

$$\frac{\frac{\Gamma, x : S \vdash N : T ! e_N}{\Gamma \vdash \lambda x.N : S \xrightarrow{e_N} T ! \emptyset} \quad \Gamma \vdash M : S ! e_M}{\Gamma \vdash (\lambda x.N)M : T ! e_N \cup e_M}$$

with $e = e_N \cup e_M$.

From this by Lemma 3 we get $\Gamma' \vdash N[M/x] : T' ! e_N \cup e_M$ where $\Gamma' = \text{push } e_M \Gamma$ and $T' = \text{push } e_M T$.

We can also generalize this, using Lemmas 1, 4 and 5, to $\Gamma'' \vdash N[M/x] : T'' ! e_N \cup e_M$ where $\Gamma'' = \text{complete } e \Gamma$ and $T'' = \text{complete } e T$.

- $\lambda x.N \rightsquigarrow \lambda x.N'$. The typing is $\Gamma \vdash \lambda x.N : S \xrightarrow{e} T ! \emptyset$ from $\Gamma, x : S \vdash N : T ! e$.

By IH, $\Gamma', x : S' \vdash N' : T' ! e$ with $\Gamma' = \text{complete } e \Gamma$, $S' = \text{complete } e S$ and $T' = \text{complete } e T$.

And so by T-ABS, $\Gamma' \vdash \lambda x.N : S' \xrightarrow{e} T' ! \emptyset$, as needed, noting $S' \xrightarrow{e} T' = \text{complete } e S \xrightarrow{e} T$. \square

Now we see that reduction preserves pure type-and-effect derivations.

Lemma 7. If $M \rightsquigarrow M'$ and $\Gamma \vdash M : T ! \emptyset$ and Γ and T are hereditarily pure then $\Gamma \vdash M' : T ! \emptyset$

Proof. A direct consequence of Lemma 6, noting also that for hereditarily-pure types T , $T = \text{complete } \emptyset T$. \square

The goal is to rewrite closed, pure terms. The next lemma shows that all such terms have their type and effect exactly preserved by the rewrite system:

Lemma 8. If $M \rightsquigarrow M'$ and $\emptyset \vdash M : T ! \emptyset$ and T is a relation type then $\emptyset \vdash M' : T ! \emptyset$. Furthermore $M \rightsquigarrow^* V$ gives then $\emptyset \vdash V : T ! \emptyset$.

Proof. A direct consequence of Lemma 6, since relation types contain no arrows and so $T = \text{complete } \emptyset T$. \square

Totality

For the rewriting to be effective, it must in fact normalize every queryzable term to a normal form in the domain of the function $\llbracket - \rrbracket$. This section shows that the system strongly normalizes: every reduction sequence terminates.

First we show that the given rewrite rules strongly normalize—that is, they admit no infinite reduction—then that the normal forms do in fact fall in the SQL-like sublanguage.

The strong normalization proof follows the pattern of the reducibility argument used by Lindley and Stark [2005], which follows the method of Tait [1967]. The proof uses “continuations” as a way of bookkeeping the progress made by certain rules, the so-called “commuting conversions,” which don’t immediately reduce the size of the term but merely commute two forms. A continuation can be thought of as a stack of contexts which need to be eliminated to normalize the term, as well as contexts that commute with those.

Definition (Continuations). Define a set of *continuations* as follows:

$$\begin{aligned} K & ::= Id \mid K \circ F \\ F & ::= (x)N \mid (\text{where } B) \mid (M \uplus) \end{aligned}$$

The notation $K @ M$ denotes filling K with the term M :

$$\begin{aligned} Id @ M & = M \\ (K \circ (x)N) @ M & = K @ (\text{for } (x \leftarrow M) N) \\ (K \circ (\text{where } B)) @ M & = K @ (\text{if } B \text{ then } M \text{ else } []) \\ (K \circ (M \uplus)) @ M' & = K @ (M \uplus M') \end{aligned}$$

Definition. The length $|K|$ of a continuation K is the number of for-abstraction frames that it contains:

$$\begin{aligned} |Id| & = 0 \\ |K \circ (x)N| & = |K| + 1 \\ |K \circ (\text{where } B)| & = |K| \\ |K \circ (M \uplus)| & = |K| \end{aligned}$$

We only count the for-abstraction frames because only they represent contexts that need to be eliminated. The others are neutral contexts—they don't react with what's placed inside them. In fact, the other frames are only present to ensure that the continuations are closed under reduction, as we will see in a moment.

The proof works by showing that terms are *reducible*, a property stronger than strong normalization, but more amenable to induction. Intuitively, a term is reducible if it strongly normalizes in every well-typed context. The reducibility predicate is indexed by the type, and at each type it is defined in terms of smaller types. For bag types $[T]$, reducibility is defined in terms of continuations, because we need to track the commuting conversions; for record and function types, there are no commuting conversions to worry about so we opt for a simpler (non-inductive) definition. Reducible base-type terms are just the strongly-normalizing ones.

Definition (Reducibility). For each type T , define a set red_T on closed terms of type T by the following rules:

- $M \in \text{red}_o$ (base type) iff M strongly normalizes,
- $L \in \text{red}_{S \rightarrow T}$ iff for every $M \in \text{red}_S$ we have that $LM \in \text{red}_T$.
- $M \in \text{red}_{(\overrightarrow{l:T})}$ iff for each $(l : T) \in (\overrightarrow{l:T})$ we have that $M.l \in \text{red}_T$
- $K \in \text{redK}_{[T]}$ iff for every $M : T$ with $M \in \text{red}_T$ we have that $K@[M]$ strongly normalizes.
- $M \in \text{red}_{[T]}$ iff for every $K \in \text{redK}_{[T]}$, $K@M$ strongly normalizes.

We will need some general properties of reducibility.

Lemma 9. There is some reducible term of every type.

Proof. A straightforward induction on the type constructs the term; we begin with a constant at base type; at type $S \rightarrow T$ we abstract a reducible N of type T with a dummy variable x to get $\lambda x.N$. For bags and records we can construct the term directly with the singleton and record constructors. \square

Next, reducibility implies strong normalization (the abbreviation “ M s.n.” is short for “ M strongly normalizes”):

Lemma 10. If $M \in \text{red}_T$ then M s.n.

Proof. By induction on T , the type of M :

- o . Immediate from the definition of reducibility.
- $S \rightarrow T$. By the definition of reducibility, $MM' \in \text{red}_T$ for each $M' \in \text{red}_S$, and then by the induction hypothesis MM' strongly normalizes. And so its subterm M strongly normalizes.

- $(\overrightarrow{l : \vec{T}})$. By definition, $M.l$ is reducible for each $l \in \vec{l}$. Therefore by the IH, $M.l$ strongly normalizes, and so its subterm M strongly normalizes.
- $[T]$. By the definition of reducibility, $Id @ M = M$ strongly normalizes. \square

Lemma 11. If $M \in \text{red}_T$ and $M \rightsquigarrow N$ then $N \in \text{red}_T$.

Proof. By induction on T , the type of M :

- Case o . If M strongly normalizes then so does N ; this suffices.
- Case $S \rightarrow T$. For each $M' \in \text{red}_S$, we have that $MM' \in \text{red}_T$ and $MM' \rightsquigarrow NM'$; then by the inductive hypothesis, $NM' \in \text{red}_T$. Since this is true for any $M' \in \text{red}_S$, this satisfies the definition of $N \in \text{red}_{S \rightarrow T}$.
- Case $(\overrightarrow{l : \vec{T}})$. For each $(l : T) \in (\overrightarrow{l : \vec{T}})$, we have that $M.l \in \text{red}_T$ and $M.l \rightsquigarrow N.l$; then by the inductive hypothesis, $N.l \in \text{red}_T$. Since this is true for each $l \in \vec{l}$, this satisfies the definition of $N \in \text{red}_{(\overrightarrow{l : \vec{T}})}$.
- Case $[T]$. For any $K \in \text{red}_{[T]}$, we have that $K @ M$ strongly normalizes; as a reduct thereof, $K @ N$ strongly normalizes. Since this is true for any $K \in \text{red}_{[T]}$, this satisfies the definition of $N \in \text{red}_{[T]}$. \square

Definition (Term contexts). A *term context* is a term with zero or more holes. A *hole* $[\]$ can stand in a context anywhere a term otherwise could. The notation $C[M]$ denotes the term formed by plugging M into *every* hole in C .

Observation. Continuations K are a subset of contexts C . The continuation plugging operation $K @ M$ is a special case of context plugging $C[M]$.

Now, we will need to speak of reductions taking place “within the continuation part” of a plugged pair like $K @ M$, so we will develop a notion of reduction for continuations. First we define reduction of contexts.

Definition (Context reduction). If $C[M] \rightsquigarrow C'[M]$ for all M then we write $C \rightsquigarrow C'$. Given a reduction $C[M] \rightsquigarrow C'[M]$ for some M we can say the reduction is “within C .”

Definition (Reduction-in-context). If we have $M \rightsquigarrow M'$ then we say that the reduction $C[M] \rightsquigarrow C[M']$ is “within M .”

Definition (Reduction at the interface). If $C[M]$ reduces and the reduction is not within C and not within M then it is “at the interface” between C and M .

Note that reductions at the interface could actually involve a deep alteration of the content M . For example, if $C = (\lambda x.C')N$ then $C[M] \rightsquigarrow C'[N/x][M[N/x]]$. This reduction is “at the interface” but can involve widespread changes in both C' and M .

Definition (Neutral term-context pairs). A term M is *neutral for the context* C if the only reductions applicable to $C[M]$ are those that are applicable within C or within M .

Since continuations are contexts, we can speak of a reduction “within K .” However, reduction on continuations is not closed: we may have $K \rightsquigarrow C'$ where C' is not a continuation. This infelicity is palliated by a lemma (N^o. 12) which produces a new and equally valuable continuation in such a case.

And also the idea of *neutrality for a given context* specializes to *neutrality for a given continuation*.

Definition (Neutral term-continuation pairs). A term M is *neutral for the continuation* K if the only reductions applicable to $K@M$ are those that are applicable within K (i.e. $K \rightsquigarrow C'$ for some C') or within M .

More generally, we want to recognize terms that are simply *neutral* toward continuations:

Definition (Neutral terms). A term is *neutral* if it is neutral for all continuations.

In inductive proofs where we take cases on the reductions of a term $K@M$, we will typically enumerate the cases for reductions *at the interface*, treating reductions in M or in K with an inductive hypothesis. The next two lemmas aid these proofs. The first shows that reductions at the interface can only involve a bounded number of continuation frames (in fact, just one), so the number of explicit cases is small. The second shows that, when a reduction is “in K ” and thus $K \rightsquigarrow C'$, we can (in a sense!) convert C' back to a continuation K' , and because of this we can safely use induction on reduction sequences.

Lemma 12. When $K \rightsquigarrow C'$, then for each term P there exists K' such that $C'[P] = K'@P$ and $|K| \geq |K'|$.

Proof. Take cases on the reduction $K \rightsquigarrow C'$.

- $K = K' \circ (x)(M \uplus N) \rightsquigarrow K'@((\text{for } (x \leftarrow [\]) M) \uplus (\text{for } (x \leftarrow [\]) N))$ (FOR-UNION-BODY)

Then

$$\begin{aligned} & (K'@((\text{for } (x \leftarrow P) M) \uplus (\text{for } (x \leftarrow P) N))) \\ &= (K' \circ ((\text{for } (x \leftarrow P) M) \uplus) \circ (x)N)@P. \end{aligned}$$

And $|K' \circ (x)(M \uplus N)| = |K' \circ ((\text{for } (x \leftarrow P) M) \uplus) \circ (x)N|$ as needed.

- $K = K' \circ (x)N \circ (y)N' \rightsquigarrow K' \circ (y)(\text{for } (x \leftarrow N') N)$ (FOR-ASSOC)

The reduct is already a continuation and

$$|K' \circ (x)N \circ (y)N'| \geq |K' \circ (y)(\text{for } (x \leftarrow N') N)|.$$

- $K = K' \circ (x)N \circ (M \uplus) \rightsquigarrow K' \circ ((\text{for } (x \leftarrow M) N) \uplus) \circ (x)N$ (FOR-UNION-SRC)

The reduct $K' \circ ((\text{for } (x \leftarrow M) N) \uplus) \circ (x)N$ is already a continuation and the length is unchanged.

- $K = K' \circ (\text{where } B) \circ (x)N \rightsquigarrow K' \circ (x)(\text{if } B \text{ then } N \text{ else } [])$. (IF-FOR)

The reduct is already a continuation and the length is unchanged.

- $K = K' \circ (x)N \circ (\text{where } B) \rightsquigarrow K' \circ (\text{where } B) \circ (x)N$. (FOR-IF-SRC)

The reduct is already a continuation and the length is unchanged.

- $K = K' \circ (\text{where } B) \circ (M\uplus) \rightsquigarrow K' \circ ((\text{if } B \text{ then } M \text{ else } [])\uplus) \circ (\text{where } B)$ (IF-UNION)

The reduct is already a continuation and the length is unchanged.

Other reductions are of the form $K = (K' \circ F) \rightsquigarrow C''[F]$. Then by the IH there is a K'' with $C''[M] = K''@M$ and $|K'| \geq |K''|$. Thence $|K' \circ F| \geq |K'' \circ F|$ as needed. \square

Now when we say that a reduction of $K@M$ is “confined to K ,” we mean that $K \rightsquigarrow C$ and that $C[M] = K'@M$ for some K' . The fact that $|K| \geq |K'|$ in this case means that we can use $|K|$ as part of an induction metric without fear that it will increase during reduction.

We will use induction on the length of the maximal reduction sequence of a term, so we introduce its notation:

Definition. Write $\text{maxred}(M)$ for the maximum length of any reduction sequence from M , defined only for strongly normalizing M .

Lemma 13. If M strongly normalizes and $M \rightsquigarrow N$, then $\text{maxred}(M) > \text{maxred}(N)$.

Proof. If the reduction is along a maximal path, then the reduct’s maximal reduction sequence is shorter by 1. If it is along some path strictly shorter than a maximal one, then the reduct’s maximal reduction sequence is also strictly shorter. \square

We will use a few general properties of reduction, as follows:

Lemma 14. If M is neutral and each reduction $M \rightsquigarrow N$ has $N \in \text{red}_T$ then $M \in \text{red}_T$.

Proof. By induction on the type T :

- o . Every reduct of M s.n. so M s.n. and $M \in \text{red}_o$ by def.
- $S \rightarrow T$. To show: that $MM' \in \text{red}_T$ for each $M' \in \text{red}_S$. Examine reductions of MM' . Because M is neutral, every reduction is within M or within M' . Every reduct of M' is reducible by Lemma 11. Thus every reduct of MM' is an application of a reducible term at $S \rightarrow T$ to a reducible term at S and is reducible. Because applications are neutral, the IH applies and $MM' \in \text{red}_T$. This satisfies the definition of $M \in \text{red}_{S \rightarrow T}$.
- $(\overline{l : \overrightarrow{T}})$. To show: that $M.l \in \text{red}_T$ for each $l : T \in \overline{l : \overrightarrow{T}}$. Examine reductions of $M.l$. Because M is neutral, every reduction is within M . Thus every reduct of $M.l$ is a projection of a reducible term and hence is reducible. Because projections themselves are neutral, the IH applies and $M.l \in \text{red}_T$. This satisfies the definition of $M \in \text{red}_{(\overline{l : \overrightarrow{T}})}$.

- $[T]$. Given $K \in \text{red}\mathbf{K}_T$ we want to show that $K@M$ s.n. Since each N for which $M \rightsquigarrow N$ has $K@N$ s.n., we only need to examine reductions in K and at the interface between K and M . But since M is neutral, there are no reductions at the interface. And since K is reducible, all its reducts are s.n. Thus all reducts of $K@M$ s.n. and hence $M \in \text{red}_T$. \square

Definition. Write $\text{size}(M)$ for the size of the term M , that is, one plus the sum of the sizes of the immediate subterms.

We proceed to prove a series of lemmas, one for each syntactic form, which assert that a term of that form is reducible when its subterms are.

Lemma 15. If $K@M$ strongly normalizes then $K@[]$ strongly normalizes.

Proof. By induction on $(|K|, \text{maxred}(K@M))$. Take cases on the reducts of $K@[]$:

- The reduct $K'@[]$, in the case where $K = K' \circ (x)N$. This strongly normalizes, by the IH.
- The reduct $K'@[]$, in the case where $K = K' \circ (\text{where } B)$. This strongly normalizes, by the IH.

All other reductions are confined to K , and so reduce $\text{maxred}(K@M)$, and thus each strongly normalizes, by the IH. \square

Lemma 16. If $M \in \text{red}_T$ then $[M] \in \text{red}_{[T]}$.

Proof. Given $K \in \text{red}\mathbf{K}_{[T]}$, we want to show that $K@[M]$ s.n. This follows directly from the definition of $\text{red}\mathbf{K}_{[T]}$ and the hypothesis $M \in \text{red}_T$. \square

Lemma 17. If $N[P/x] \in \text{red}_T$ for every $P \in \text{red}_S$ then $\lambda x.N \in \text{red}_{S \rightarrow T}$.

Proof. Given $P \in \text{red}_S$, we want to show that $(\lambda x.N)P$ strongly normalizes. This follows by induction on $\text{maxred}(N) + \text{maxred}(P)$. Take cases on the term's reducts:

- $(\lambda x.N)P \rightsquigarrow N[P/x] \in \text{red}_T$ by hypothesis.

All other reductions are confined to N or P , preserving the respective reducibility property, by Lemma 11, and these reductions decrease the induction metric. \square

Lemma 18. Given $(\overrightarrow{l = \vec{M}}) : (\overrightarrow{l : \vec{T}})$, if $M_l \in \text{red}_{T_l}$ for each $l \in \vec{l}$ then $(\overrightarrow{l = \vec{M}}) \in \text{red}_{(\overrightarrow{l : \vec{T}})}$.

Proof. By induction on $\sum_{M \in \vec{M}} \text{maxred}(M)$. We show that for any field $l \in \vec{l}$ with type T_l , the projection $(\overrightarrow{l = \vec{M}}).l \in \text{red}_{T_l}$. Examine the reducts. Centrally, $(\overrightarrow{l = \vec{M}}).l \rightsquigarrow M_l$ (by RECORD- β) which by hypothesis is in red_{T_l} . Any other reductions are confined to one of the \vec{M} and so the induction hypothesis applies. \square

Lemma 19. Given a continuation K and terms M and N , if $K@M$ and $K@N$ strongly normalize then $K@(M \uplus N)$ strongly normalizes.

Proof. By lexicographic induction on $(|K|, \text{maxred}(K@M) + \text{maxred}(K@N))$. For the base case, when $K = Id$, the s.n. of $M \uplus N$ follows directly from that of M and N . Now take cases on the reducts of $K@(M \uplus N)$:

- $K@(M \uplus N) \rightsquigarrow K'@((\text{for } (x \leftarrow M) L) \uplus (\text{for } (x \leftarrow N) L)),$ (FOR-UNION-SRC)
when $K = K' \circ (x)L$.

The application $K'@(\text{for } (x \leftarrow M) L) \equiv K@M$ strongly normalizes by hypothesis and likewise does $K@N$. By IH, then, $K'@((\text{for } (x \leftarrow M) L) \uplus (\text{for } (x \leftarrow N) L))$ s.n. (IH applies because $|K| > |K'|$.)

- $K@(M \uplus N) \rightsquigarrow K'@((\text{if } B \text{ then } M \text{ else } []) \uplus (\text{if } B \text{ then } N \text{ else } [])),$ (IF-UNION)
when $K = K' \circ (\text{where } B)$.

The same reasoning applies as in the first case.

All other reductions are confined to K , M or N , thus reducing the induction metric. \square

Lemma 20. If $K@(M \uplus N)$ strongly normalizes, then $K@M$ and $K@N$ strongly normalize, and

$$\begin{aligned} \text{maxred}(K@(M \uplus N)) &\geq \text{maxred}(K@M), \text{ and} \\ \text{maxred}(K@(M \uplus N)) &\geq \text{maxred}(K@N). \end{aligned}$$

Proof. Any reduction in $K@M$ has a corresponding nonempty reduction sequence in $K@(M \uplus N)$ which produces a new term of the form $K'@(M' \uplus N')$. Therefore no reduction sequence of $K@M$ is longer than the maximal one of $K@(M \uplus N)$ and in particular $K@M$ must strongly normalize.

We construct the corresponding sequence as follows. A reduction in $K@M$ must be in K , in M , or at the interface. If it is in K or in M , the reduction applies directly to $K@(M \uplus N)$. If it is at the interface, then $K@M = (K' \circ F)@M \rightsquigarrow K'@M'$. Take cases the reduction $F@M \rightsquigarrow M'$:

- Case FOR-UNION-SRC; we have $F@M = \text{for } (x \leftarrow M) N' \rightsquigarrow M'$.

$$\begin{aligned} &(K' \circ (x)N')@(M \uplus N) \\ \rightsquigarrow &K'@((\text{for } (x \leftarrow M) N') \uplus (\text{for } (x \leftarrow N) N')) \\ \rightsquigarrow &K'@(M' \uplus (\text{for } (x \leftarrow N) N')) \end{aligned}$$

- Case IF-UNION; we have $F@M = \text{if } B \text{ then } M \text{ else } [] \rightsquigarrow M'$

$$\begin{aligned} &(K' \circ (\text{where } B))@(M \uplus N) \\ \rightsquigarrow &K'@((\text{if } B \text{ then } M \text{ else } []) \uplus (\text{if } B \text{ then } N \text{ else } [])) \\ \rightsquigarrow &(K'@(M' \uplus (\text{if } B \text{ then } N \text{ else } []))). \end{aligned}$$

A symmetrical argument applies for $K@N$. \square

Lemma 21. If $M \in \text{red}_{[T]}$ and $N \in \text{red}_{[T]}$ then $M \uplus N \in \text{red}_{[T]}$.

Proof. Immediate from Lemma 19. \square

Lemma 22. Given a continuation $K : [T]$, a strongly normalizing term $L : S$, and a frame $(x)N : [S] \rightarrow [T]$, if $K@(N[L/x])$ strongly normalizes then $K@(\text{for } (x \leftarrow [L])N)$ strongly normalizes.

Proof. By lexicographic induction on

$$(|K|, \text{maxred}(K@(N[L/x]) + \text{maxred}(L), \text{size}(N))).$$

Take cases on the reducts of $K@(\text{for } (x \leftarrow [L])N)$, to show that each strongly normalizes:

- $K@(\text{for } (x \leftarrow [L]) N) \rightsquigarrow K@(N[L/x])$ (FOR- β)

This s.n. by hypothesis.

- $K@(\text{for } (x \leftarrow [L]) N) \rightsquigarrow K@[]$ (FOR-ZERO-BODY)
if $N = []$.

This s.n. by Lemma 15. The precondition that $K@x$ strongly normalizes follows from the strong normalization of $K@(N[L/x])$.

- $K@(\text{for } (x \leftarrow [L]) N) \rightsquigarrow K'@(\text{for } (x \leftarrow [L]) (\text{for } (y \leftarrow N) M)),$ (FOR-ASSOC)
when $K = K' \circ (y)M,$
with $x \notin \text{FV}(M).$

By the ind. hyp., using K' and $\text{for } (y \leftarrow N) M$ for K and N , resp., we get that $K@(\text{for } (x \leftarrow [L]) \text{for } (y \leftarrow N) M)$ s.n., as required. To see that the IH applies, note that

$$K'@((\text{for } (y \leftarrow N) M)[L/x]) = K@(N[L/x])$$

by hypothesis (remembering $x \notin \text{FV}(M)$), and $|K| > |K'|$.

- $K@(\text{for } (x \leftarrow [L]) N) \rightsquigarrow K@((\text{for } (x \leftarrow [L]) N_1) \uplus (\text{for } (x \leftarrow [L]) N_2))$ (FOR-UNION-BODY)
if $N = N_1 \uplus N_2.$

Because $K@(N_1 \uplus N_2)[L/x]$ s.n. we know $K@N_1[L/x]$ and $K@N_2[L/x]$ s.n. (Lemma 20).

Then by the IH, $K@(\text{for } (x \leftarrow [L]) N_1)$ and $K@(\text{for } (x \leftarrow [L]) N_2)$ s.n. (IH applies because the size of N decreases and $|K|$ and the maximal-reduction component do not increase.) Finally, by Lemma 19, we conclude that

$$K@((\text{for } (x \leftarrow [L]) N_1) \uplus (\text{for } (x \leftarrow [L]) N_2)) \text{ s.n.}$$

- $K@(\text{for } (x \leftarrow [L]) N) \rightsquigarrow K'@(\text{for } (x \leftarrow [L]) (\text{if } B \text{ then } N \text{ else } [])),$ (IF-FOR)
if $K = K' \circ (\text{where } B).$

We have that $x \notin \text{FV}(B)$ by the assumption that all bound variables are distinct.

Therefore

$$\begin{aligned} K@N[L/x] &= K'@((\text{if } B \text{ then } N \text{ else } [])[L/x]) \\ &= K'@(\text{if } B \text{ then } N[L/x] \text{ else } []) \end{aligned}$$

and this strongly normalizes by hypothesis.

Then by the IH, $K'@(\text{for } (x \leftarrow [L]) (\text{if } B \text{ then } N \text{ else } []))$ s.n. IH applies because $|K| > |K'|$.

Any other reduction takes place within K , N or L , so the claim for those cases follows by the IH, reducing the induction metric. \square

Lemma 23 (Reducibility of for terms). If $M \in \text{red}_{[S]}$ and N is such that $N[L/x] \in \text{red}_{[T]}$ for any $L \in \text{red}_S$ then $\text{for } (x \leftarrow M) N \in \text{red}_{[T]}$.

Proof. Given any $K \in \text{redK}_{[T]}$ we need to show that $K@(\text{for } (x \leftarrow M) N)$ s.n. By hyp., for any $K' \in \text{redK}_{[S]}$, we know that $K'@M$ s.n. Pick $K' = K \circ (x)N$ to show this condition, that $K' \in \text{redK}_{[S]}$. To show this requires showing that $K'@[L]$, equivalently $K@(\text{for } (x \leftarrow [L]) N)$, strongly normalizes, for any reducible L . By Lemma 22 and hypotheses, this is the case. So $K'@M$ strongly normalizes, which is what we wanted to write. \square

Definition. Define Q -contexts as follows:

$$Q ::= [] \mid \text{for } (x \leftarrow Q) M \mid \text{for } (x \leftarrow M) Q \mid M \uplus Q \mid \text{if } B \text{ then } Q \text{ else } []$$

Definition. Define a measure $|Q|$ as follows:

$$\begin{aligned} |[]| &= 0 \\ |\text{for } (x \leftarrow Q) M| &= 1 + |Q| \\ |\text{for } (x \leftarrow M) Q| &= |Q| \\ |M \uplus Q| &= |Q| \\ |\text{if } B \text{ then } Q \text{ else } []| &= |Q| \end{aligned}$$

Reduction of Q -contexts is just a case of reduction of contexts, so $Q \rightsquigarrow C'$ iff $Q@M \rightsquigarrow C'[M]$. As with K -continuations, we need to be able to refit a Q -context, when the reduction duplicates the hole, to leave a single hole.

Lemma 24. Whenever $Q \rightsquigarrow C'$ we have for each P some Q' such that $C'[P] = Q'@P$ with $|Q| \geq |Q'|$.

Proof attempt. By cases on the reduction.

- $\text{for } (y \leftarrow M \uplus M') Q \rightsquigarrow (\text{for } (y \leftarrow M) Q) \uplus (\text{for } (y \leftarrow M') Q)$. (FOR-UNION-SRC)

Given P we have

$$\begin{aligned} & ((\text{for } (y \leftarrow M) Q) \uplus (\text{for } (y \leftarrow M') Q))[P] = \\ & ((\text{for } (y \leftarrow M) (Q[P])) \uplus (\text{for } (y \leftarrow M') Q))@P, \end{aligned}$$

with the part before the @ a Q -context.

Meanwhile

$$|\text{for } (y \leftarrow M \uplus M') Q| = |Q| = |(\text{for } (y \leftarrow M) (Q[P])) \uplus (\text{for } (y \leftarrow M') Q)| = |Q|.$$

- $\text{for } (y \leftarrow Q) (M \uplus M') \rightsquigarrow (\text{for } (y \leftarrow Q) M) \uplus (\text{for } (y \leftarrow Q) M')$ (FOR-UNION-BODY)

Given P we have

$$\begin{aligned} & ((\text{for } (y \leftarrow Q) M) \uplus (\text{for } (y \leftarrow Q) M'))[P] = \\ & ((\text{for } (y \leftarrow (Q[P])) M) \uplus (\text{for } (y \leftarrow Q) M'))@P, \end{aligned}$$

with the part before the @ a Q -context.

Meanwhile

$$|\text{for } (y \leftarrow Q) M \uplus M'| = 1 + |Q| = |(\text{for } (y \leftarrow (Q[P])) M) \uplus (\text{for } (y \leftarrow Q) M')|.$$

- $\text{for } (y \leftarrow M \uplus Q) M' \rightsquigarrow \text{for } (y \leftarrow M) M' \uplus \text{for } (y \leftarrow Q) M'$ (FOR-UNION-SRC)

Similar to previous cases.

- $\text{for } (y \leftarrow M) (M' \uplus Q) \rightsquigarrow \text{for } (y \leftarrow M) M' \uplus \text{for } (y \leftarrow M) Q$ (FOR-UNION-BODY)

Similar to previous cases.

- $\text{if } B \text{ then for } (x \leftarrow Q) N \text{ else } [] \rightsquigarrow \text{for } (x \leftarrow Q) (\text{if } B \text{ then } N \text{ else } [])$ (IF-FOR)

The reduct is a Q -context and

$$|\text{if } B \text{ then for } (x \leftarrow Q) N \text{ else } []| = 1 + |Q| = |\text{for } (x \leftarrow Q) (\text{if } B \text{ then } N \text{ else } [])|.$$

- $\text{if } B \text{ then } (\text{for } (x \leftarrow M) Q) \text{ else } [] \rightsquigarrow \text{for } (x \leftarrow M) (\text{if } B \text{ then } Q \text{ else } [])$ (IF-FOR)

The reduct is a Q -context and

$$|\text{if } B \text{ then for } (x \leftarrow M) Q \text{ else } []| = |Q| = |\text{for } (x \leftarrow M) (\text{if } B \text{ then } Q \text{ else } [])|.$$

- $\text{for } (x \leftarrow \text{if } B \text{ then } Q \text{ else } []) M \rightsquigarrow$ (FOR-IF-SRC)
 $\text{if } B \text{ then } (\text{for } (x \leftarrow Q) M) \text{ else } []$

The reduct is a Q -context and

$$|\text{for } (x \leftarrow \text{if } B \text{ then } Q \text{ else } []) M| = 1 + |Q| = |\text{if } B \text{ then } (\text{for } (x \leftarrow Q) M) \text{ else } []|.$$

- $\text{for } (x \leftarrow \text{if } B \text{ then } M \text{ else } []) Q \rightsquigarrow$ (FOR-IF-SRC)
 $\text{if } B \text{ then } (\text{for } (x \leftarrow M) Q) \text{ else } []$

The reduct is a Q -context and

$$|\text{for } (x \leftarrow \text{if } B \text{ then } M \text{ else } []) Q| = |Q| = |\text{if } B \text{ then } (\text{for } (x \leftarrow M) Q) \text{ else } []|.$$

- $\text{for } (x \leftarrow \text{for } (y \leftarrow Q) N') N \rightsquigarrow$ (FOR-ASSOC)
 $\text{for } (y \leftarrow Q) \text{for } (x \leftarrow N') N$

The reduct is a Q -context and

$$|\text{for } (x \leftarrow \text{for } (y \leftarrow Q) N') N| = 2 + |Q| \geq |\text{for } (y \leftarrow Q) \text{for } (x \leftarrow N') N| = 1 + |Q|.$$

- $\text{for } (x \leftarrow \text{for } (y \leftarrow M') Q) N \rightsquigarrow$ (FOR-ASSOC)
 $\text{for } (y \leftarrow M') \text{for } (x \leftarrow Q) N$

The reduct is a Q -context and

$$|\text{for } (x \leftarrow \text{for } (y \leftarrow M') Q) N| = 1 + |Q| = |\text{for } (y \leftarrow M') \text{for } (x \leftarrow Q) N|.$$

- $\text{for } (x \leftarrow \text{for } (y \leftarrow M) N) Q \rightsquigarrow$ (FOR-ASSOC)
 $\text{for } (y \leftarrow M) \text{for } (x \leftarrow N) Q$

The reduct is a Q -context and

$$|\text{for } (x \leftarrow \text{for } (y \leftarrow M) N) Q| = |Q| = |\text{for } (y \leftarrow M) \text{for } (x \leftarrow N) Q| = |Q|.$$

- $\text{if } B \text{ then } M \uplus Q \text{ else } [] \rightsquigarrow$ (IF-UNION)
 $\text{if } B \text{ then } M \text{ else } [] \uplus \text{if } B \text{ then } Q \text{ else } []$.

The reduct is a Q -context and

$$|\text{if } B \text{ then } M \uplus Q \text{ else } []| = |Q| = |\text{if } B \text{ then } M \text{ else } [] \uplus \text{if } B \text{ then } Q \text{ else } []|.$$

End of proof attempt.

[TBD: FOR- β reductions can come from deep in Q without being in Q .]

← !

Lemma 25. If $Q@M$ s.n. then $Q@[]$ s.n.

Proof. By lexicographic induction on $(|Q|, \text{maxred}(Q@M))$. Take cases on the reductions of $Q@[]$:

- $(Q' \circ (\text{for } (x \leftarrow M) []))@[] \rightsquigarrow Q'@[]$ (FOR-ZERO-BODY)
S.n. by IH.
- $(Q' \circ (\text{for } (x \leftarrow [] M)))@[] \rightsquigarrow Q'@[]$ (FOR-ZERO-SRC)
S.n. by IH.
- $(Q' \circ (\text{if } B \text{ then } [] \text{ else } []))@[] \rightsquigarrow Q'@[]$ (IF-ZERO)
S.n. by IH.

Other reductions are within Q and so reduce $\text{maxred}(Q@M)$. \square

Lemma 26. Given a Q-context Q and terms M and N , if $Q@M$ and $Q@N$ strongly normalize then $Q@(M \uplus N)$ strongly normalizes.

No proof yet. [Similar to Lemma 19]

Lemma 27. If $Q@(M \uplus N)$ strongly normalizes, then $Q@M$ and $Q@N$ strongly normalize, and

$$\begin{aligned} \text{maxred}(Q@(M \uplus N)) &\geq \text{maxred}(Q@M), \text{ and} \\ \text{maxred}(Q@(M \uplus N)) &\geq \text{maxred}(Q@N). \end{aligned}$$

No proof yet. [Similar to Lemma 20]

Lemma 28. If $Q@N$ s.n. and $\text{FV}(B) \cap \text{BV}(Q) = \emptyset$ then $Q@(\text{if } B \text{ then } N \text{ else } [])$ s.n.

Proof attempt. By induction on $(\text{maxred}(Q@N) + \text{maxred}(B), |Q|, \text{size}(N))$. Take cases on the reductions of $Q@(\text{if } B \text{ then } N \text{ else } [])$:

- $Q@(\text{if } B \text{ then } (\text{for } (x \leftarrow M') N') \text{ else } [])) \rightsquigarrow$ (IF-FOR)
 $Q@(\text{for } (x \leftarrow M') (\text{if } B \text{ then } N' \text{ else } []))$ if $N = \text{for } (x \leftarrow M') N'$.

The reduct s.n. by the inductive hypothesis, letting Q be $Q \circ (\text{for } (x \leftarrow M') [])$ and N be N' . We have $\text{size}(N) > \text{size}(N')$ and the length of the new Q is $|Q \circ (\text{for } (x \leftarrow M') [])|$ while $Q@N$ is unchanged. We have $x \notin \text{FV}(B)$ by the assumption of distinct binders. *huzzah!*

- $Q'@(\text{for } (x \leftarrow [L]) (\text{if } B \text{ then } N \text{ else } [])) \rightsquigarrow$ (FOR- β)
 $Q'@((\text{if } B \text{ then } N \text{ else } [])[L/x])$ if $Q = Q' \circ \text{for } (x \leftarrow [L]) []$.

Because $x \notin \text{FV}(B)$ we have

$$\begin{aligned} Q'@((\text{if } B \text{ then } N \text{ else } [])[L/x]) &= \\ Q'@(\text{if } B \text{ then } N[L/x] \text{ else } []) & \end{aligned}$$

so we show the latter is s.n.

This follows by the inductive hypothesis, letting Q be Q' and N be $N[L/x]$. We have $\text{maxred}(Q'@(\text{for } (x \leftarrow [L]) N)) > \text{maxred}(Q'@N[L/x])$.

- $Q'@(for (x \leftarrow if B then N else []) M) \rightsquigarrow$ (FOR-IF-SRC)
 $Q'@(if B then (for (x \leftarrow N) M) else [])$ if $Q = Q' \circ for (x \leftarrow []) M$.

The reduct s.n. by the inductive hypothesis, letting Q be Q' and $for (x \leftarrow N) M$. We have $Q@N$ unchanged while $|Q| > |Q'|$.

- $Q@if B then N else [] \rightsquigarrow$ (IF-ZERO)
 $Q@[]$ if $N = []$.

The reduct s.n. by Lemma 25.

- $Q@(if B then (N_1 \uplus N_2) else []) \rightsquigarrow$ (IF-UNION)
 $Q@((if B then N_1 else []) \uplus (if B then N_2 else []))$ if $N = N_1 \uplus N_2$.

By the inductive hypothesis, we have that $Q@(if B then N_1 else [])$ s.n. and likewise $Q@(if B then N_2 else [])$. To see that IH applies, note that Q is unchanged, $size(N_1) < size(N) > size(N_2)$ and (by Lemma 27)

$$maxred(Q@N_1) \leq maxred(Q@N) \geq maxred(Q@N_2).$$

Then Lemma 26 combines these two facts to show that

$$Q@((if B then N_1 else []) \uplus (if B then N_2 else [])) \text{ s.n.}$$

All other reductions are within Q , B or N and so reduce $maxred(Q@N) + maxred(B)$.
End of proof attempt.

Lemma 29. If B s.n. and $K@M$ s.n., then $K@(if B then M else [])$ s.n.

Proof. The set of continuations K is a subset of the continuation-contexts Q , and in particular $BV(K) = \emptyset$. Thus the lemma follows immediately from Lemma 28. \square

Lemma 30. Given a continuation $K : [T]$ and terms $B : \text{bool}$, $M : [T]$ and $N : [T]$, if B , $K@M$ and $K@N$ strongly normalizes, then $K@(if B then M else N)$ strongly normalizes

Proof. By induction on $maxred(B) + maxred(K@M) + maxred(K@N)$. When $N = []$. Lemma 29 applies. So consider the case $N \neq []$. Take cases on the reducts of $K@(if B then M else N)$.

- The reduct $K@(if B then M else [] \uplus if \neg B then N else [])$ (IF-SPLIT)
if $N \neq []$.

Now $K@M$ and $K@N$ s.n. by hyp., so by Lemma 29, $K@(if B then M else [])$ s.n. and likewise $K@(if \neg B then N else [])$. Then using Lemma 19 we combine these two facts to show that the reduct s.n.

Other reductions are confined to B , M , N or K , and so reduce the induction metric, and hence the reducts are s.n. by IH. \square

Lemma 31 (Reducibility of conditionals at bag type). If $M \in \text{red}_{[T]}$ and $N \in \text{red}_{[T]}$ and B strongly normalizes then $\text{if } B \text{ then } M \text{ else } N \in \text{red}_{[T]}$.

Proof. Immediate from Lemma 30. \square

At last we can show that conditionals of any type are reducible.

Lemma 32 (Reducibility of conditionals at any type). If $M, N \in \text{red}_T$ and B s.n. then $(\text{if } B \text{ then } M \text{ else } N) \in \text{red}_T$.

Proof. By induction on T .

- Case $[T]$. By Lemma 31.
- Case $S \rightarrow T$. Given a term M' reducible at type S , we must show that the application $(\text{if } B \text{ then } M \text{ else } N)M'$ is reducible at T . We show that all its reducts are reducible. This is by induction on $\text{maxred}(B) + \text{maxred}(M) + \text{maxred}(N) + \text{maxred}(M')$. Consider the reductions. First

$$(\text{if } B \text{ then } M \text{ else } N)M' \rightsquigarrow \text{if } B \text{ then } MM' \text{ else } NM' : T.$$

By (outer) IH, this reduct is in red_T .

All other reductions are confined to B , M , N , or M' , producing a term that is reducible by the (inner) IH.

Because all its reducts are reducible, and applications are neutral, by Lemma 14, $(\text{if } B \text{ then } M \text{ else } N)M'$ and then also $\text{if } B \text{ then } M \text{ else } N$ are reducible.

- Case $(\overrightarrow{l : T})$.

We must show that for any $(l : T_l) \in (\overrightarrow{l : T})$, the projection $(\text{if } B \text{ then } M \text{ else } N).l$ is reducible at T_l . We show that all its reducts are reducible. This is by induction on $\text{maxred}(B) + \text{maxred}(M) + \text{maxred}(N)$. Consider the reductions. First,

$$(\text{if } B \text{ then } M \text{ else } N).l \rightsquigarrow (\overrightarrow{l = L}).l \quad (\text{IF-RECORD})$$

where for each l , $L_l = \text{if } B \text{ then } M.l \text{ else } N.l$.

By (outer) IH, each L_l is in red_{T_l} , and so by Lemma 18, the construction $(\overrightarrow{l = L})$ is in $\text{red}_{(\overrightarrow{l : T})}$. By definition, this means that $(\overrightarrow{l = L}).l$ is in red_{T_l} .

All other reductions are confined to B , M , N , producing a term that is reducible by the (inner) IH.

Because all its reducts are reducible, and projections are neutral, we have by Lemma 14 that $(\text{if } B \text{ then } M \text{ else } N).l$ and hence $\text{if } B \text{ then } M \text{ else } N$ are reducible.

- Case o . None of B , M or N can react with its context; hence all reductions are confined to B , M or N and strong normalization follows directly from the hypotheses. \square

Lemma 33. If $M \in \text{red}_T$ then $\text{empty}(M) \in \text{red}_{\text{bool}}$.

Proof. We merely need to show that $\text{empty}(M)$ strongly normalizes. If M is of relation type then no reductions apply except for those applying within M , which strongly normalizes. Any normal form V of M has the same type and thus we have a normal form $\text{empty}(V)$ of $\text{empty}(M)$.

If M is not of relation type then we apply induction on $\text{maxred}(M)$. Consider the reduct $\text{empty}(M) \rightsquigarrow \text{empty}(\text{for } (x \leftarrow M) [()])$. Now $\text{for } (x \leftarrow M) [()]$ strongly normalizes by virtue of $M \in \text{red}_T$ (letting $K = \text{Id} \circ (x) [()]$). For any normal form V of $\text{for } (x \leftarrow M) [()]$, we know that $\text{empty}(V)$ is a normal form, because V is relation type. Any other reduction is within M and the goal follows by the IH. \square

Proposition 1 (Reducibility of closed terms). Given any typed term $y_1 : S_1, \dots, y_n : S_n \vdash M : T$ and corresponding closed terms $L_1 \in \text{red}_{S_1}, \dots, L_n \in \text{red}_{S_n}$, we have $M[\overrightarrow{L/y}] \in \text{red}_T$.

Proof. By induction on the structure of M .

- c . Trivially strongly normalizing, hence (because they have base type) reducible.
- $\text{table } s : T$. Already normal; trivial.
- y_i . Each free variable of M is substituted with a reducible term; trivial.
- $[]$. Follows immediately from Lemma 15.
- $[M]$. Let T be the type of M . By ind. hyp., $M[\overrightarrow{L/y}]$ is in red_T , and then by Lemma 16, $[M][\overrightarrow{L/y}]$ is in $\text{red}_{[T]}$.
- $\text{for } (x \leftarrow M) N$. Let $[T]$ be the type of M and $[S]$ be the type of N . We can assume $x \notin \vec{y}$, using α -conversion as necessary.

By ind. hyp., $M[\overrightarrow{L/y}]$ is in $\text{red}_{[T]}$ and

$$N[\overrightarrow{L/y}, P/x] = N[\overrightarrow{L/y}][P/x] \in \text{red}_{[S]}$$

for each $P \in \text{red}_T$ (recall the \vec{L} are closed). Then by Lemma 23, $(\text{for } (x \leftarrow M) N)[\overrightarrow{L/y}]$ is in $\text{red}_{[S]}$.

- $M \uplus N$.
By ind. hyp., $M[\overrightarrow{L/y}]$ and $N[\overrightarrow{L/y}]$ are in $\text{red}_{[T]}$; then by Lemma 21, $(M \uplus N)[\overrightarrow{L/y}]$ is, too.
- $MN : T$. By ind. hyp., we have $M[\overrightarrow{L/y}] \in \text{red}_{S \rightarrow T}$ and $N[\overrightarrow{L/y}] \in \text{red}_S$; together these directly imply $MN \in \text{red}_T$.

- $\lambda x.N : S \rightarrow T$.

By ind. hyp., we have that all closing substitutions on N give a reducible term, so in particular $N[\overrightarrow{L/y}][P/x] \in \text{red}_T$ for any $P \in \text{red}_S$. Then by Lemma 17 we get that $\lambda x.N[\overrightarrow{L/y}] = (\lambda x.N)[\overrightarrow{L/y}] \in \text{red}_{S \rightarrow T}$,

- $M.l$. By IH, $M \in \text{red}_{(\overrightarrow{l/T})}$, which implies directly that well-formed $M.l \in \text{red}_T$.

- $(\overrightarrow{l = M})$. By Lemma 18.

- if B then M else N .

By IH, $M[\overrightarrow{L/y}]$ and $N[\overrightarrow{L/y}]$ are in red_T , $B[\overrightarrow{L/y}] \in \text{red}_B$; then by Lemma 32, $(\text{if } B \text{ then } M \text{ else } N)[\overrightarrow{L/y}] \in \text{red}_T$.

- $\text{empty}(M)$. By IH, $M \in \text{red}_T$ and so by Lemma 33 we get $\text{empty}(M) \in \text{red}_{\text{bool}}$.

- query M . Taking cases on the reducts, which are either just $M[\overrightarrow{L/y}]$ or produced by making reductions within $M[\overrightarrow{L/y}]$, their strong normalization follows directly by the inductive hypothesis. \square

Lemma 34 (Unsubstitution). If $M[\overrightarrow{L/y}]$ strongly normalizes then M does.

Proof. By induction on M ; most cases are simple applications of the induction hypothesis. For the base case of a variable, x , the variable strongly normalizes regardless of the substitution, so we're done. \square

Proposition 2. Any well-typed term strongly normalizes.

Proof. Given M which may have free variables \vec{x} , there is some well-typed substitution $[\vec{L}/\vec{x}]$ which produces a closed term $M[\vec{L}/\vec{x}]$, and by the reducibility of closed terms this is reducible. Reducible terms are strongly normalizing, so $M[\vec{L}/\vec{x}]$ strongly normalizes. Then the Unsubstitution Lemma shows that M itself strongly normalizes. \square

Effect analysis

Lemma 35. If $\Gamma \vdash M : T ! e$ is a valid typing derivation, and no abstractions appear anywhere in M , then every subexpression N with typing derivation $\Gamma' \vdash N : S ! e'$ has $e' \subseteq e$.

Proof. The typing rule T-ABS is the only one that permits an immediate subderivation to include effects that do not appear in the consequent. So if abstractions do not appear in the term, this rule is not used and every subexpression's effects are contained within the whole expression's effects. \square

Normal forms

Proposition 3. Closed, well-typed terms of effect-free relation type have normal forms that satisfy this grammar:

$$\begin{array}{ll}
\text{(normal forms)} & V, U, W ::= V \uplus U \mid [] \mid F \\
\text{(comprehension NFs)} & F ::= \text{for } (x \leftarrow \text{table } s : T) F \mid Z \\
\text{(comprehension bodies)} & Z ::= \text{if } B \text{ then } Z \text{ else } [] \mid [R] \mid \text{table } s : T \\
\text{(row forms)} & R ::= (\overrightarrow{l = \vec{B}}) \mid x \\
\text{(basic expressions)} & B ::= \text{if } B \text{ then } B' \text{ else } B'' \mid \text{empty}(V) \mid \\
& \quad \oplus(\vec{B}) \mid x.l \mid x \mid c
\end{array}$$

To prove the special case of closed, effect-free relation-type terms, we first characterize *all* the normal forms.

Lemma 36 (Normal forms). Every well-typed \rightsquigarrow -normal form falls in this grammar:

$$\begin{array}{ll}
\text{(normal forms)} & V, U, W ::= V \uplus U \mid [] \mid F \mid R \\
\text{(comprehension NFs)} & F ::= \text{for } (x \leftarrow L) F \mid Z \\
\text{(table-like forms)} & L ::= \text{table } s : T \mid B \\
\text{(comprehension bodies)} & Z ::= \text{if } I \text{ then } Z \text{ else } [] \mid [V] \mid L \\
\text{(nonbag expressions)} & R ::= (\overrightarrow{l = \vec{V}}) \mid I \\
\text{(nonbag, nonrecord expr'ns)} & I ::= \text{if } I_1 \text{ then } I_2 \text{ else } I_3 \mid \lambda x.V \mid B \\
\text{(basic forms)} & B ::= BV \mid B.l \mid x \mid c \mid \oplus(\vec{V}) \mid \\
& \quad \text{empty}(V)
\end{array}$$

Proof. The proof shows, by induction on the structure of terms, that each term is either ill-typed, non-normal, or matches the above grammar.

Observe that the nonterminal V encompasses all the others in this grammar. As a result we can take the IH as asserting that subterms match the grammar V .

- $M \uplus N$, $[]$, $[M]$, $\text{table } s : T$, $(\overrightarrow{l = \vec{M}})$, $\lambda x.N$, x , c . These meet the grammar, applying the inductive hypothesis where subterms are concerned.
- $\text{for } (x \leftarrow L) M$. Take cases on L which by IH meets the grammar V :
 - $V \uplus U$, $[]$, $[V]$, $\text{for } (x \leftarrow L) F$ and $\text{if } B \text{ then } Z \text{ else } []$. Active for the context.
 - $\text{if } I_1 \text{ then } I_2 \text{ else } I_3$ with $I_3 \neq []$. This term is either bag-typed, and rewrites by IF-SPLIT, or else is not bag-typed and cannot be well-typed in this context.
 - $\text{table } s : T$, BV , $B.l$, x , c . These meet the grammar.
 - $\lambda x.N$, $(\overrightarrow{l = \vec{M}})$. Ill-typed in this context.

Take cases on M , which by IH meets the grammar V :

- $V \uplus U$, $[]$. Active for the context.

- terms matching F . These meet the grammar.
- $(\overrightarrow{l = M})$, $\lambda x.N$. Ill-typed in this context.
- if I_1 then I_2 else I_3 with $I_3 \neq []$. If this term is bag-typed, it rewrites by IF-SPLIT; otherwise it is ill-typed in this context.

- if M' then M else N .

The condition M' must be of type `bool`, and so must be a constant, a deconstruction, or a variable, hence it matches I .

Now take cases on whether the type of `if M' then M else N` is a bag type, a record type, or some other.

If it has bag type and N is not `[]` then the term rewrites. So consider the case that N is `[]`. Take cases on M , which by IH must match the grammar V :

- $V \uplus U$, `[]`, for $(x \leftarrow L) F$. Active for the context.
- terms matching Z . Meets the grammar.
- terms matching P . Ill-typed in this context

If it has record type, it rewrites.

If it does not have bag or record type, take cases on M and N , which must match the grammar V ; we enumerate one set of cases since M and N are treated symmetrically:

- $V \uplus U$, `[]`, $(\overrightarrow{l = V})$, F . Ill-typed in this context.
- Terms matching I . Meet the grammar.

- LM . Take cases on L , which must meet the grammar V :

- $V \uplus U$, `[]`, F . Ill-typed in this context.
- $(\overrightarrow{l = V})$. Ill-typed in this context.
- if I_1 then I_2 else I_3 , $\lambda x.N$. Active for the context.
- `:` Terms matching B . Meet the grammar.

- $\oplus(\overrightarrow{V})$. Meets the grammar.

- `empty(V)`. Meets the grammar.

- `query(V)`. Rewrites to V , hence non-normal.

- $V.l$. The only normal form of record type is the record construction $V = (\overrightarrow{l = V})$; but then $V.l$ forms a redex and is not in normal form, a contradiction. \square

Next we tighten the grammar for the normal forms of relation-type terms.

First we need a lemma that shows that basic forms B essentially inherit their type from the environment—this is because basic forms consist only of a series of destructors applied to a variable.

Definition. A type S is a *subformula* of a type T , written $S \curlywedge T$, iff S appears within T ; the relation is the least one satisfying these laws:

$$\begin{aligned}
& S \curlywedge S \\
& S \curlywedge T \implies S \curlywedge T' \xrightarrow{e} T \\
& S \curlywedge T \implies S \curlywedge T \xrightarrow{e} T' \\
& S \curlywedge T \implies S \curlywedge (l : T, \overrightarrow{l : T'}) \\
& S \curlywedge T \implies S \curlywedge [T]
\end{aligned}$$

Lemma 37. For any effect-free basic form B with typing $\Gamma \vdash B : T ! \emptyset$ the type T is either a base type or a subformula of the type of one of the variables in Γ .

Proof. By induction on the structure of B . Each syntactic form of B either has base type, or is a deconstructor of a B form, or is a variable. By cases:

- Case BV . Here B must have type $S \rightarrow T$. By the IH, B has hereditary typing wrt Γ .
- Case $B.l$. Here B must have type $(\overrightarrow{l : S})$ with $(l : T) \in (\overrightarrow{l : T'})$, and by IH B has hereditary typing wrt Γ .
- Case x . Here $x : T$ is in Γ .
- Case c . Constants have base type.
- Case $\oplus(\overrightarrow{V})$. By prescription, since this is effect-free it has base type.
- Case $\text{empty}(V)$. Has base type. □

Now in an environment that assigns a row type to each variable, we can more tightly characterize the possible forms, as follows.

Lemma 38. Given any relation-type normal form V with typing $\Gamma \vdash V : T$ where Γ assigns row type to each x in $\text{dom}(\Gamma)$, we have that all free and bound variables appearing in V have row type, and for any subterm for $(x \leftarrow L) Z$ we have that L is of the form $\text{table } s : [(\overrightarrow{l : \circ})]$.

Proof. By induction on the structure of V :

- Case for $(y \leftarrow L) Z$.

Any free variables appearing in L come from the environment Γ and thus have row type. Thus L cannot itself be a variable. By an inductive argument, it cannot

have either of the forms BV or $B.l$ since this would require variables with function type or whose type is a record with a bag field. Finally L cannot be a constant since constants are assumed to have base type. Thus L can only have the form $\text{tables} : [(\overrightarrow{l : \delta})]$. By the inductive hypothesis, extending Γ with a binding $y : (\overrightarrow{l : \delta})$, we get the result for subterms in Z .

- Cases if I then Z else $[], [V], \text{table } s : T, BV$ and $B.l$. Here the inductive hypothesis directly gives our proposition.
- Case x . By hypothesis, x has row type. □

Lemma 39. If Γ gives a row type to each variable and B is normal and effect-free with typing $\Gamma \vdash B : T ! \emptyset$ then B is not an application form $B'V$.

Proof. If B had the form $B'V$ then B' would have a type $S \rightarrow T$ and this $S \rightarrow T$ would have to be a subformula of one of the types in Γ (by Lemma 37), yet these are all row types, a contradiction. □

Lemma 40. If Γ gives a row type to each variable and B is normal and effect-free with typing $\Gamma \vdash B : T ! \emptyset$ with T a base type then B cannot have the form $B'V$ or x .

Proof. Suppose B has the form $B'V$; then B' has type of the form $S \rightarrow T$ and $S \rightarrow T$ is a subformula of one of the types in Γ (by Lemma 37), yet these are all row-type, a contradiction. Suppose B is a variable, x . Then its type appears in Γ ; yet x has base type and all the variables in Γ have row type, a contradiction. □

Lemma 41. If Γ gives a row type to each variable and B is normal and effect-free with typing $\Gamma \vdash B : T ! \emptyset$ with T a row type then B cannot have the form $B'V, B'.l, c, \text{empty}(V)$ or $\oplus(\vec{V})$.

Proof. The forms $c, \text{empty}(V)$ and $\oplus(\vec{V})$ all have base type and so would be ill-typed. If B has the form $B'V$, the term B' has type of the form $S \rightarrow T$, and this is a subformula of one of the types in Γ (by Lemma 37), yet these are all row-type, a contradiction. If B has the form $B'.l$ then the field l of B' must have a row type, making B' something strictly larger than a row type; and as such it cannot be a subformula of one of the types in Γ , a contradiction. □

Lemma 42. If Γ gives a row type to each variable and Z is normal with typing $\Gamma \vdash Z : T$ with T a relation type, then Z cannot have any of the forms ranged by B , and if it has the form $[V]$ then $V = (\overrightarrow{l = \vec{I}})$ and each field member I has one of the forms if B_1 then B_2 else $B_3, \text{empty}(V), \oplus(\vec{B}), x.l, x$ or c .

Proof. Suppose Z falls in the set ranged by B . Then its type must be a subformula of one of the types in Γ (by Lemma 37). But since Z has relation-type and none of the elements of Γ can have such a subformula, this is a contradiction.

Now suppose Z is a singleton list $[V]$. Because it is relation-typed, V must be row-typed. Thus it cannot have any of the forms $V \uplus U, [], F$ (recalling that it cannot

have any form ranged by B). This leaves the forms ranged by R . At row type, it cannot have a conditional form since this is not normal. It cannot be an abstraction because this would be ill-typed. Thus it can only be a record construction $(\overline{l = \vec{V}})$ with each V_i having base type. The only normal forms which have base type are those ranged by B and the form $\text{if } I_1 \text{ then } I_2 \text{ else } I_3$. Within the forms ranged by B , Lemma 39 shows that it cannot have the form $B'V$. And it cannot have be a variable x because this would give it relation type in the context Γ . If it has the form $B'.l$, then B' can only be a variable (other possibilities being ill-typed due to the hereditary typing or given conditions). Remaining are the forms $\text{if } B_1 \text{ then } B_2 \text{ else } B_3$, $\text{empty}(V)$, $\oplus(\vec{B})$, $x.l$, x and c as we wanted to show. \square

Lemma 43. Wherever $\text{empty}(V)$ appears in a normal-form term, V has relation type.

Proof. The rule `EMPTY-FLATTEN` ensures this. \square

Observation. Each type/effect inference rule other than that for functional abstractions $\lambda x.N$ is monotonic in its effects. That is, if the conclusion is $\Gamma \vdash M : T ! e$ then each precondition $\Gamma' \vdash M' : T' ! e'$ has $e' \subseteq e$.

Corollary. If a term has no functional abstractions, then its entire derivation is monotonic in the effects. That is, for each derivation to $\Gamma \vdash M : T ! e$, each of its subderivations, to $\Gamma' \vdash M' : T' ! e'$, has $e' \subseteq e$.

Lemma 44. In an effect-free normal form of relation type, any operation application $\oplus(\vec{V})$ has all its arguments of base type, thus each argument meets the grammar for B .

Proof. Because it is effect-free, the term contains no abstractions, and so the typing derivation is monotonic in its effects. By the side condition that every primitive either has arguments all of base type or has an effect, we can infer that primitives in an effect-free term with no abstractions have arguments all of base type, and thus meet the normal-form grammar for B . \square

At last we can prove the full characterization of the normal forms of query-bracketed terms.

Proof of Prop. 3. Striking from the grammar of Lemma 36 the forms disallowed by Lemmas 38, 39, 40, 41, 42, 43, and 44 we are left with a grammar for the normal forms of closed relation-type expressions:

$$\begin{array}{ll}
\text{(normal forms)} & V, U, W ::= V \uplus U \mid [] \mid F \\
\text{(comprehension NFs)} & F ::= \text{for } (x \leftarrow \text{table } s : T) F \mid Z \\
\text{(comprehension bodies)} & Z ::= \text{if } B \text{ then } Z \text{ else } [] \mid [R] \mid \text{table } s : T \\
\text{(record expressions)} & R ::= (\overline{l = \vec{B}}) \mid x \\
\text{(basic expressions)} & B ::= \text{if } B \text{ then } B' \text{ else } B'' \mid \text{empty}(V) \mid \\
& \oplus(\vec{B}) \mid x.l \mid c
\end{array}$$

Which is what we wanted to show. \square

4 Adding Recursion

A general-purpose programming language without recursion would be severely hobbled; but in standard SQL, recursive queries are not expressible. Thus we need to add recursion to our language but ban it from query expressions.

We add recursion by introducing a recursive λ -abstraction, spelled `recfun`. It introduces a recursive function of one argument and forces the resulting function type to have a `noqy` effect.

$$\frac{\Gamma, f : S \xrightarrow{e \cup \{\text{noqy}\}} T, x : S \vdash M : T ! e}{\Gamma \vdash \text{recfun } f \ x = M : S \xrightarrow{e \cup \{\text{noqy}\}} T ! \emptyset} \quad (\text{T-RECFUN})$$

Alternatively, we could introduce a fixpoint operator:

$$\frac{\Gamma \vdash M : (S \xrightarrow{e} T) \xrightarrow{\emptyset} (S \xrightarrow{e} T) ! \emptyset}{\Gamma \vdash \text{fix } M : S \xrightarrow{e \cup \{\text{noqy}\}} T ! \emptyset} \quad (\text{T-FIX})$$

This prohibition is conservative—it forbids all primitive-recursive programs, for example, although many are translatable to SQL—but reasonable, since programmers are not in the habit of writing queries that require that much power.

5 Adding the length operator

The examples in Section ?? used a function `length` which we have not yet studied. This section shows how to extend the system to support it. In the source, it is much like `empty`, but SQL’s nonuniformity forces us to handle it specially.

First we extend the source language with `length`:

$$M ::= \dots \mid \text{length}(M)$$

The typing rule is as you would expect, resulting in type `int`.

Extend the SQL-like sublanguage (the normal forms) as follows:

$$B ::= \dots \mid \text{length}(F)$$

Note that we will normalize the argument to a comprehension normal form, F , so that it gives a `select` query and not, say, a `union all` query. Next, augment the SQL target:

$$e ::= \dots \mid \text{select count}(\ast) \text{ from } \overrightarrow{t \text{ as } x} \text{ where } e$$

And hence we can translate it to SQL as follows:

$$\llbracket \text{length}(F) \rrbracket = \text{select count}(\ast) \text{ from } \overrightarrow{t \text{ as } x} \text{ where } e \\ \text{where select } \overrightarrow{s} \text{ from } \overrightarrow{t \text{ as } x} \text{ where } e = \llbracket F \rrbracket$$

Now we add the following rewrite rules:

$$\begin{aligned}
\text{length}(M) : T &\rightsquigarrow \text{length}(\text{for } (x \leftarrow M) [()]) && \text{(LENGTH-FLATTEN)} \\
&&& \text{if } M \text{ is not relation-typed} \\
\text{length}([]) : T &\rightsquigarrow 0 && \text{(LENGTH-ZERO)} \\
\text{length}(M \uplus N) : T &\rightsquigarrow \text{length}(M) + \text{length}(N) && \text{(LENGTH-UNION)}
\end{aligned}$$

Thus assumes, of course, that we have the constant 0 and the integer-addition operation (+) in our set of constants and primitives.

Related Work

Exploring the relaxation of the 1st-normal-form in database theory (the restriction that every relation must be flat), Paredaens and Van Gucht [Paredaens and Van Gucht, 1988, 1992] first showed that nested relational algebra expressions can be converted to flat relational algebra expressions. Their algebra involved only relational expressions and not functional abstraction, thus it is a first-order algebra.

Limsoon Wong and Leonid Libkin later generalized this work in a series of papers about Nested Relational Calculus (NRC), which permits functional abstraction but only in application position (such as $(\lambda x.N)M$). This work showed that expressions in NRC need not construct intermediate data structures that are more deeply nested than the maximum of the tables and the query result.

The definition of NRC and the effort to harmonize it with programming languages goes back at least to “Naturally Embedded Query Languages” [Breazu-Tannen et al., 1992] and “Comprehension Syntax” [Buneman et al., 1994].

The query system Kleisli [Wong, 2000] is structured as a general-purpose programming language, which compiles (sometimes partially) into SQL. Those parts of a Kleisli program that the compiler cannot translate are executed directly by the interpreter; there is no *a priori* way to determine what programs, or parts thereof, would compile to SQL.

The LINQ project [Microsoft Corporation, 2005] is a set of extensions to the .NET framework which allow expressing database queries (targeting SQL, XML and other data models) in two ways: through an object interface or through SQL-like syntactic sugar. The object interface provides methods for query operations, such as mapping, filtering, and so on. Many of these methods accept code, in the form of “expression trees.” For instance, the **Where** method, for filtering, takes as argument a predicate with which to filter. Up to a point, this facility permits arbitrary code from the host language to be added to queries. But not all code is successfully translated. For example, it is possible to use an abstract predicate as a query condition; but it is not possible to compose abstract predicates. This is because functions for use in queries have a distinct type, `Expr<Function<A,B>>`, rather than `Function<A,B>`, and the former does not afford composition in a way that the query translator will recognize. So, for example if *pred* is an abstracted predicate of type `Expr<Function<B,Bool>>` and *f* is an abstracted

function of type $\text{Expr}\langle\text{Function}\langle A, B \rangle\rangle$, and one wishes to filter a bag of rows to those rows x such that $\text{pred}(f(x))$ is true, it is necessary to declare a new function $\text{pred}2$ which implements the composition.

It bears noting that LINQ, like Kleisli, allows the expression of queries that can't be expressed in SQL, but whose results can be constructed from an SQL query. For example, a LINQ query can give its result rows in the form of an object type, which has no direct analogue in SQL; in this case the query generator may still perform the bulk of the query in SQL and simply repackage the results during a post-processing phase. Such a splitting is not afforded by the system of this report, because this system is based in a hard assertion of what code must translate to SQL. It may be desirable to find a flexible middle ground which would allow expressions to be split, as in LINQ and Kleisli, and still offers some static guarantees of queryization, as in this chapter.

Hillebrand et al. [1993] prove an inverse result to our present one, that simply-typed λ -calculus itself subsumes other query languages (flat relational calculus, Datalog⁻, and others), and in particular that PTIME queries become λ -terms evaluable in PTIME.

Wiedermann and Cook [2007] present a technique using abstract interpretation for extracting structured queries from program code. Its source language is imperative and first-order: it has constructs for assignment, sequencing, record projection, primitives, conditions, and iteration over a collection, but not for function abstraction or application. The analysis produces queries in OQL, the Object Query Language.

Fegaras [1998] also gives an unnesting rewrite system for a nested relational calculus that differs from this one in that it generalizes the bag comprehension to any monoid comprehension, but does not include side-effects or recursion. Fegaras proves the soundness but not the totality of this algorithm; the present paper suggests a technique for showing completeness by way of strong normalization and effect analysis.

Other database systems that apply comprehensions for querying include the QLC query language of Erlang's Mnesia [Mattsson et al., 1998], LINQ, and Links.

History

The science of language-integrated query begins with Schek and Scholl [1986], who first defined an algebra of nested relations, thus freeing queries from the flatland of the 1st-normal-form restriction. This was the first step towards integrating databases with the spaceland of general-purpose programming, where orthogonality is prized over normalization, and where data structures are limited only by imagination.

Paredaens and Van Gucht [1988, 1992] made the first step towards the present result by showing that nested relational algebra, when restricted to flat input and output relations, is equivalent to traditional flat relational algebra. Wong [1996] soon extended this result, showing that any nested relational algebra expression can be rewritten so that it produces no intermediate data structures deeper than the greatest of its input and output relations.

This body of theory bore practical fruit first in Kleisli [Buneman et al., 1995, Wong, 2000], a functional-programming system that allows querying flat relational databases,

as well as other data sources, and allowing the use of nested relations as intermediate values and as results.

The LINQ project [Microsoft Corporation, 2005] integrates queries into several programming languages, using an object-oriented interface for forming queries. The interface has multiple implementations, for querying both in-memory data structures, directly, and databases, using SQL—so the same source queries can draw from either sort of data source. Query expressions, including predicates and data transformations, can be written in the host language; a linguistic *quoting* mechanism captures the code itself for use at runtime, when a translator will traverse this code looking for fragments it can translate to SQL; if it can, it does so, producing a query as well as possible pre- and post-processing phases that were untranslatable.

The Links project [Cooper et al., 2006] also offers language-integrated query. Like in Kleisli, the queries are expressed using the language’s own ordinary iteration constructs and conditionals. In the published version of Links, as with the other systems, there is no sure way to predict how much code will be translated into queries, and how much will be performed outside the database.

The study of language-integrated query is still young.

Close language integration for queries has fundamental limitations, since general-purpose programming languages are more expressive than SQL. Besides the problem of nested data structures, programming languages normally have operations that cannot be expressed in SQL, including recursion, side-effecting statements, and primitive functions that simply aren’t available.

In LINQ, for example, query objects may contain arbitrary expressions from the host language. The tension is resolved by providing no guarantee on whether a query object will entirely become a query. An arbitrary amount of work may be left to the pre- and post-processing phases, or the query translator may simply throw an exception at runtime.

In Links, the translator’s limitations never cause an expression to die; at worst, Links will simply fetch all the data from the database and do all the processing itself. In the original version of Links, there is no way to predict or control what the translator will do with an expression, and this is an unhappy prospect, given DBMSes’ good indexing and query planning facilities.

Kleisli, too, offers no guarantees about what code will be translated. It faces a different challenge from the others, because it allows expressing queries across several separate data sources, so there is no one way to divide the code between them.

References

- Val Breazu-Tannen, Peter Buneman, and Limsoon Wong. Naturally embedded query languages. In *ICDT '92*. Springer, 1992.
- P. Buneman, S. B. Davidson, K. Hart, C. Overton, and L. Wong. A data transformation system for biological data sources. In *VLDB '95*, pages 158–169, 1995.

- Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. Comprehension syntax. *SIGMOD Record*, 23:87–96, 1994.
- Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. In *FMCO '06*, 2006.
- Leonidas Fegaras. Query unnesting in object-oriented databases. In *SIGMOD '98*, pages 49–60, New York, NY, USA, 1998. ACM.
- Torsten Grust. *The Functional Approach to Data Management*, chapter Monad comprehensions, a versatile representation for queries. Springer Verlag, 2003.
- Gerd G Hillebrand, Paris C Kanellakis, and Harry G Mairson. Database query languages embedded in the typed lambda calculus. In *LICS '93*, 1993.
- Sam Lindley and Ian Stark. Reducibility and $\top\top$ -lifting for computation types. In *TLCA '05*, pages 262–277, 2005.
- Håkan Mattsson, Hans Nilsson, and Claes Wikström. Mnesia—a distributed robust DBMS for telecommunications applications. In *PADL '99*, pages 152–163. Springer, 1998.
- Microsoft Corporation. The LINQ project: .NET language integrated query. White paper, September 2005.
- Jan Paredaens and Dirk Van Gucht. Possibilities and limitations of using flat operators in nested algebra expressions. In *PODS '88*, pages 29–38, New York, NY, USA, 1988. ACM.
- Jan Paredaens and Dirk Van Gucht. Converting nested algebra expressions into flat algebra expressions. *ACM Trans. Database Syst.*, 17(1):65–93, 1992.
- H. J. Schek and M. H. Scholl. The relational model with relation-valued attributes. *Inf. Syst.*, 11(2):137–147, 1986.
- W. W. Tait. Intensional interpretation of functionals of finite type I. *Journal of Symbolic Logic*, 32(2):198–212, 1967.
- Jean-pierre Talpin and Pierre Jouvelot. The type and effect discipline. In *Information and Computation*, pages 162–173, 1992.
- Phil Trinder and Philip L. Wadler. List comprehensions and the relational calculus. In *Glasgow Workshop on Functional Programming*, pages 187–202, 1989. URL citeseer.ist.psu.edu/wadler99list.html.
- Ben Wiedermann and William R. Cook. Extracting queries by static analysis of transparent persistence. In *POPL '07*, 2007.

Limsoon Wong. Kleisli, a functional query system. *J. Functional Programming*, 10(1): 19–56, January 2000.

Limsoon Wong. Normal forms and conservative extension properties for query languages over collection types. *J. Comput. Syst. Sci.*, 52(3):495–505, 1996.